AD-A228 580

PENGUIN: A Language for Reactive Graphical User Interfaces

Sue-Ken Yap

Technical Report 344
April 1990

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

90

PENGUIN: A Language for Reactive Graphical User Interfaces

by

葉仕堅 (Sue-Ken Yap)

Submitted in Partial Fulfillment

of the

Requirements for the Degree

DOCTOR OF PHILOSOPHY

Supervised by Michael Scott

Department of Computer Science

University of Rochester

Rochester, New York

April 1990

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 344	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) PENGUIN: A Language for Reactive Graphical User Interfaces		5. TYPE OF REPORT & PERIOD COVERED technical report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Sue-Ken Yap		8. CONTRACT OR GRANT NUMBER(s) N00014-82-K-0193
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Dept., 734 Computer Studies Bldg. University of Rochester, Rochester, NY 14627		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd., Arlington, VA 22209		12. REPORT DATE April 1990
		13. NUMBER OF PAGES 115
14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office) Office of Naval Research, Inf. Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

user interfaces; human factors; interaction techniques; grammars; parsing

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(see reverse)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT

PENGUIN (Programming Environment for Graphical User Interfaces) is a computer language that supports grammar-based specification of control flow in event-driven graphical programs. The PENGUIN model of intercomponent connection extends and subsumes the older Seeheim model of UIMS design, allowing large programs to be constructed as co-operating components.

Traditional approaches to graphical program design have treated the user interface code as something that can be written independently of the application. This dissertation argues that the reactive nature of graphical programs should be taken into account from the beginning of design, that a graphical program be composed as a collection of modules whose input behaviour is specified, and that modules be grouped into separately-compiled components along lines of clear division of labour and responsibility for resources. Such partitions result in components that are more likely to be reusable.

This dissertation discusses the rationale behind the design of PENGUIN. An implementation of PENGUIN was used to evaluate this design. Our experiences indicate that the use of PENGUIN can reduce the volume of user interface code by a factor of two to three and result in code which is clearer than functionally equivalent code using traditional control structures. Uniform handling of I/O and signals as PENGUIN events leads to programs that are more portable across systems.

PENGUIN encourages the construction of *reactive* modules which are clearer than modules that use *polling* or a mixed approach. We claim that reactive programming is a sounder basis for constructing the sophisticated interfaces that will eventually supplant the current generation of interaction techniques.

Curriculum Vitae

葉仕堅 (Yap Sue Ken) was born in Kuala Lumpur, Malaysia. He received his Bachelor of Electrical Engineering from the University of Malaya in 1979. During his engineering studies, he became interested in the potential of computers. Upon graduation, he went to the Basser Department of Computer Science of the University of Sydney, Australia. He received the M.Sc. degree in 1982. He worked for two years in industry as a systems analyst before returning to graduate school. Since 1984 he has been enrolled in the Computer Science Ph.D. program at the University of Rochester, NY, USA.

Acknowledgments

The single authorship of this thesis belies significant contributions by other people. My family, especially my mother, have been very understanding to let me go to the other side of the world for six years. Without their implicit support, I could not have finished. The Department of Computer Science supported me financially throughout the duration of the doctorate program. My mentor and advisor, Michael Scott, has been very patient with me throughout my candidacy. My colleagues Laura Sanchis and Michael Swain were kind enough to proofread early drafts of this thesis. Finally, my thanks to the staff of Carlson Library, one of the best stocked libraries I have used, and especially to Diane Reiman, who understands what computers are really for.

This material is based upon work supported in part by the National Science Foundation under Grant number CCR-8320136 and by ONR/DARPA research contract number N00014-82-K-0193. The government has certain rights in this material.

Abstract

PENGUIN (Programming Environment for Graphical User Interfaces) is a computer language that supports grammar-based specification of control flow in event-driven graphical programs. The PENGUIN model of intercomponent connection extends and subsumes the older Seeheim model of UIMS design, allowing large programs to be constructed as co-operating components.

Traditional approaches to graphical program design have treated the user interface code as something that can be written independently of the application. This dissertation argues that the reactive nature of graphical programs should be taken into account from the beginning of design, that a graphical program be composed as a collection of modules whose input behaviour is specified, and that modules be grouped into separately-compiled components along lines of clear division of labour and responsibility for resources. Such partitions result in components that are more likely to be reusable.

This dissertation discusses the rationale behind the design of PENGUIN. An implementation of PENGUIN was used to evaluate this design. Our experiences indicate that the use of PENGUIN can reduce the volume of user interface code by a factor of two to three and result in code which is clearer than functionally equivalent code using traditional control structures. Uniform handling of I/O and signals as PENGUIN events leads to programs that are more portable across systems.

PENGUIN encourages the construction of *reactive* modules which are clearer than modules that use *polling* or a mixed approach. We claim that reactive programming is a sounder basis for constructing the sophisticated interfaces that will eventually supplant the current generation of interaction techniques.

Table of Contents

List of Figures

1 Introduction

This chapter explains the scope of this dissertation, motivates its goals, defines terminology, and outlines the organization of the remainder of the text.

1.1 Overview

This dissertation is concerned with the design of a notation and computer language, PENGUIN, for specifying and programming *interactive graphical user interfaces*, and a model for the composition of graphics programs.

Graphical user interfaces (GUIs) make computer applications easier for non-technical users to use, but because of their size and complexity these interfaces require more effort in program construction than textual interfaces do. Current techniques of constructing GUI software are inadequate because they are based on an inadequate model of program structure and input/output handing, and fail to address the particular needs of graphical interfaces. An examination of the requirements shows that formal notations, together with automated tools for code generation, can provide a solution to this programming bottleneck.

The thesis of this dissertation is (1) that graphical user interfaces require linguistic support for event-driven programming, grammar-based specification of input and composition of independent processes, and (2) that PENGUIN meets these needs.

PENGUIN has these characteristics:

- The input language of PENGUIN programs is specified with an augmented context-free grammar. The grammar notation supports two goals: it enhances the readability of specifications, and it allows the automatic generation of parsers that enforce the input language.
- PENGUIN programs are *reactive* in contrast to conventional programs that either poll for input or alternate between program control and input control.

- A *module* is the PENGUIN unit of compilation and the basic building block for creating interactive programs. It contains a grammar specification, private data, and executable code. A module embodies an interaction technique or simulates a physical object in software. Modules are composed into components at compile time.

- A *component* is the PENGUIN unit of independent, concurrent execution. PENGUIN generalizes the three-layer Seeheim model [Pfaff, 1985] to the PENGUIN model which allows arbitrary graphs of components. Components can be composed dynamically to create networks that work in concert. PENGUIN provides precise definitions for compatibility between components. Components that have compatible protocol specifications can be connected together. Components that have the same protocol specification are interchangeable as far as their peers are concerned.

1.2 Motivation

Computers are powerful tools. The history of computing machinery has seen technology drive increases in computing power and memory capacity, with no foreseeable barriers to the current rate of improvement.

Human-computer interfaces have improved tremendously at the same time. Today's GUIs are a far cry from the patchboards of early machines. Unfortunately we do not have the software technology for constructing lucid and maintainable programs for today's graphics hardware.

1.2.1 Reactive execution

One significant problem is the traditional model of input/output (I/O) in standard programming languages. I/O is accessed via routines from a software library (as in C [Kernighan and Ritchie, 1978]) or via constructs built into the language (as in Pascal [Jensen and Wirth, 1975]). This model of I/O is not suitable for GUIs for several reasons. With GUIs there can be more than one focus of attention. The user has the freedom to enter information at various places and in no particular order. I/O subroutines cannot handle this mode of interaction easily. Systems like the MacIntosh [Apple Computers, 1986] or the X Windowing System [Scheifler and Gettys, 1987] that provide access to graphics devices from conventional programming languages either turn the program into a single threaded state machine or require the use of non-standard routines that poll for input to avoid blocking at inconvenient places.

Consider the implementation of a typesetting previewer. The previewer displays a page, then reads a user command (keystroke or mouse button depression)

to direct its next action. In addition, the previewer has to respond to signals from the windowing system that indicate the need to repaint the current window contents. In the normal programming paradigm, the program is written as a "big loop with switch" thus:

```
display a page
loop
    get an input event
    case event of
        keystroke:
        mouse button:
            do user command
        repaint signal:
            repaint window
    /* end of cases */
/* end of loop */
```

The first problem with the code above is that it is not easily discerned that it describes several interactive threads, nor is the input language of each thread clear from the program code. Errors may be made in writing a program to conform to input specifications. It may also be difficult to change or rewrite the code with assurance that the input behaviour has not been altered. Suppose the programmer decides to implement multiple character commands. An example would be a command to jump to a certain page, which requires collecting the digits of a number. If the digits are read within the case branch, then repaint signals are ignored until the whole number has been collected. Immediate response to repaint signals can be preserved only by incorporating the number reading routine in the main loop, thereby adversely affecting program clarity.

PENGUIN's grammar-based specification keeps unrelated threads of interaction lexically separated. The sequence of events leading to a particular action can be more readily deduced from the PENGUIN specification than from the "big loop with switch" code. Equally important, unrelated code managing separate resources can be kept lexically separate in modules, and not be mixed together in the big loop.

A side benefit of programming with a high level model of program input is that the portability of the application is improved because the specification deals with abstract events, pushing details of the implementation to environment-specific libraries. Also the error messages that the user receives from the GUI compiler will reflect constructs in the design abstraction and not the constructs of the target environment.

A second problem with the code above comes to light if the programmer wants to modify the program so that the user can interrupt the display of a page. The fetching and display of individual characters of the page must be incorporated into the main display loop of the program if it is to remain sensitive to interrupts while the page is partially complete. Alternatively, system-specific means of handling interrupts must be used. In short, the current paradigm permits no more than a single loop if the reactive nature of the program is to be preserved.

An event driven approach to GUI programming removes many of these difficulties. Unfortunately, current languages do not support event driven programming well. A subroutine package such as the X11 library does not sufficiently motivate the programmer to write programs in an event-driven style; self-discipline is required of the programmer to preserve the event-driven nature of the program with current tools. PENGUIN's event-driven model frees the programmer from the contortions required to translate algorithms in the event-driven world to a sequential implementation.

1.2.2 Inter-process connections

Current User Interface Management System (UIMS) models have inflexible views of graphics programs. A program is usually designed to work in one particular environment and do one task. Co-operation with other programs is either difficult or impossible. The traditional dichotomy between the user interface and the application code results in programs that cannot communicate with each other, only with the the shared graphical display software. A re-examination of the rationale behind the traditional UIMS model is needed.

PENGUIN defines a model for composing programs as a network of communicating components. This model is a generalization of the traditional Seeheim model, discussed in Chapter 2, which partitions a graphical program into three components according to logical function. PENGUIN allows the program to be partitioned into an arbitrary number of components according to division of labour. The PENGUIN model results in components that are better delineated in terms of input protocol and management of private resources, and therefore more likely to be interchangeable or reusable. Since PENGUIN specifies the behaviour of each component with a grammar, the interactive behaviour of the network of connected components is also specified.

1.3 Goals

A language and model for programming GUIs should meet the following goals:

Specification It should be usable as a specification language. Programmers who are reading code written in this notation should be able to understand the intention of the author better than if the interface had been programmed in a standard programming language.

Translation A compiler can generate code from the specification. The compiler should significantly reduce the amount of coding effort required to construct an interface compared with current methods.

Portability The notation should be sufficiently independent of the graphics environment (the underlying operating system support and the graphics libraries available) that the effort of porting a program to a different environment is small compared to the effort of rewriting the program, had it been written without the aid of the notation.

Reusability The language should encourage the reuse of code by making it easy for the programmer to keep the portions that can be used in another program, while altering the portions that must be customized for the new application. Modules like sliders, buttons, menus, etc. should be available for easy reuse.

1.4 Contributions

- PENGUIN is a specification language for GUIs. Programmers can specify the language accepted by the program with an augmented context-free grammar. This makes the task of program maintenance easier and improves the portability of the program by helping the programmer separate the specification of the protocol of the program from the implementation.

- PENGUIN explores the view that interactive programs are inherently event-driven and execution should be driven by the arrival of data. Traditional programming has considered I/O an activity subordinate to the main work of the program. This dissertation argues that an input-driven model [van den Bos, 1988] is more natural for graphics programs.

- In conventional I/O, multiple threads of interaction are usually merged into a loop that dispatches each event to the appropriate subroutine. This obfuscates the code by juxtaposing code for unrelated threads of activity. In PENGUIN unrelated sequences are lexically separated. PENGUIN exploits the event-driven model to support an efficient implementation of multiple threads.

- PENGUIN is an implementation language for GUIs. A PENGUIN compiler translates the high level specification into the target language. This reduces

the amount of coding required of a programmer by implementing a higher level of abstraction. PENGUIN also reduces the amount of effort required to port large graphics programs by concentrating environment dependencies in the run time support.

- PENGUIN modules encourage the reuse of interface code. A module is a unit of interaction and is a realization of an interaction method. A module is a self-contained manager of a resource. Multiple instances of a module can be created, all sharing the same behaviour but each with its own copy of variables. Libraries of modules are available as off-the-shelf components. Variant modules can be derived from existing modules instead of being written from scratch.

- PENGUIN proposes a replacement for the traditional Seeheim model that better fits the needs of programs, and results in a more symmetric and uniform treatment of the co-operating components of a program. Such components are more likely to be interchangeable or reusable.

1.5 Definitions

This dissertation will heavily use, both for illustrative purposes, and for validation of the methodology, hardware configurations commonly known as workstations with windowing displays. The methodology can easily be extended to deal with advanced interactive devices which are coming into more widespread use. However, workstations are currently the most accessible representatives of a class of computing environments in which programs have to deal with interactive, interleaved inputs from multiple sources and where graphical information, which is much richer in attributes than textual information, is processed. A brief explanation of the terminology involved follows.

1.5.1 Hardware

A *bitmap display* is a screen whose unit of addressability is the *pixel*, short for *picture element*, in contrast to textual displays, where the unit is the character. A pixel may be monochrome, gray scale or colour. The ability to display arbitrary patterns of pixels gives great flexibility to programs. Text and graphics are handled by the same general-purpose mechanism. As an example, the character font is no longer restricted to that provided by the hardware, as in textual displays, but can be controlled at will by software.

1.5.2 Windowing Systems

A bitmap display is too precious a resource to devote entirely to a single program. A *windowing system* is software that partitions the "real-estate" of the screen into separate regions, commonly called *windows*, each of which can be assigned to a different program. The windowing system also multiplexes the use of the input devices, such as the keyboard and mouse, between the windows. The use of the display can be further increased by allowing windows to overlap. Windows in use obscure those kept idle. This is called the *desktop metaphor*, for the analogy to the desk of a busy person.

Note that although a window may be a text window, and commonly is, a window is simply a delineated region of the screen[1]. For example, a chessboard may be presented in a window. In the X Windowing System [Scheifler and Gettys, 1987], windows may enclose subwindows ad infinitum, which is pleasantly regular. The chessboard may use subwindows for the squares.

1.5.3 Physical and Logical Devices

Each window on a physical display can be considered a logical output device. The mouse and keyboard appear to be devoted exclusively to the window selected for input. In other words, each window is associated with one or more logical input devices. The windowing system, together with a *window manager*, determines the placement of windows and the mapping of physical I/O devices to logical I/O devices.

Foley and Wallace [Foley and Wallace, 1974] classify input devices by function. For example:

- pick, e.g. lightpen or mouse
- button
- locator, e.g. joystick, tablet or mouse
- valuator, e.g. dial or slider

A device from one class can be emulated by a device from a different class, albeit with reduced effectiveness. For example, a typical graphical implementation of a volume control is as a simulated slider controlled by the mouse.

To the program, only the range and resolution of data returned by a device are of interest. The device is treated as a black box that returns some information

[1]Windows are usually rectangular so that the computations to multiplex input and direct output can be efficient, but this is not a requirement.

about the real world. The implementation, in terms of physical devices, perhaps via layers of logical devices, may affect the ergonomics of usage, and concerns only the designer of the device. In this dissertation we shall consider only logical devices. The multiplexing of physical devices onto logical ones and the emulation of one logical device by another are outside the scope of this dissertation.

1.5.4 People Involved

In this text, the *user* is a person who operates the program to achieve some goal. Such a person is not assumed to know anything technical about computers. The *programmer* is a person who implements the program. Knowledge of the application program, and some knowledge of the principles of user interface design is expected of a programmer. The *designer* is a person who creates software tools and libraries such as PENGUIN that enable programmers to build programs with high-quality GUIs.

1.6 Organization

The rest of this dissertation is organized as follows: Chapter 2 summarizes the state of the art and explains why conventional programming paradigms are inadequate. In chapter 3 we propose a model of program construction and a grammar-based notation for expressing the multi-threaded nature of such programs. We explore the technical issues in the design of a practical language, PENGUIN, that embodies these ideas and the algorithms needed to implement this language in chapter 4. Chapter 5 introduces and justifies the PENGUIN model for the construction of large programs. In chapter 7 we relate experiences with using PENGUIN. Finally, in chapter 8, we discuss the lessons learnt and how these can applied to future work in this area.

2 Previous Work

In this chapter we survey previous work in characterizing and formalizing GUI specifications, and models for composing GUIs.

2.1 Characteristics of Graphical Interfaces

Programming languages provide facilities for input/output. However, the I/O constructs in these languages are oriented toward textual interfaces. I/O facilities that are suitable for textual interfaces may be inadequate for GUIs. Several significant differences between textual and graphical user interfaces can be enumerated.

Multiple attributes Textual interfaces use elements of information drawn from a small, predetermined alphabet, generally the same alphabet used in written human communication. The alphabets of GUIs can be drawn from many diverse sets, e.g. size, value, texture, colour, orientation, shape, intensity, time, symbols and so forth [Bertin, 1983; Morse, 1979]. Some of these alphabets are open-ended sets--symbols, for example. Some alphabets may be composed from a small, recognizable set. For an alphabet of symbols, the primitives are lines and geometrical figures.

Interleaved interaction Textual interfaces usually assume that the computer and human alternate at sending information. Typically, the computer generates some output and waits for the user to enter some data. GUIs can be operated with both machine and user working concurrently.

Multiple interaction sites Textual interfaces usually assume that there is just one focus of attention and that interaction between human and machine takes place at that location. Although the screen is two dimensional, the textual interaction can be characterized by a linear string of symbols from human and machine. GUIs can have many interaction locations. The user is not constrained to finish an interaction at one place before moving to the

next place. The user is free to interact with the machine in any sensible sequence. This means that the origin of the input data is just as important as the contents of the data and should be made available to the program.

2.2 The Seeheim Model

Early workers visualized the human/computer interface as a part of a program that could be separately designed and fitted into existing programs. Implicit in the term User Interface Management System (UIMS) was the idea that the interface could be installed as a piece of software through which all interactions with programs were channeled, by analogy with Data Base Management Systems.

A structural model of GUIs was first proposed at the SIGGRAPH workshop on Graphical Input Interaction (GIIT) in 1982 [Thomas, 1983]. The following year the workshop in User Interface Management Systems in Seeheim [Pfaff, 1985] produced the *Seeheim Model* of interactive application structure.

The Seeheim Model has three layers—the application, the dialogue and the presentation.

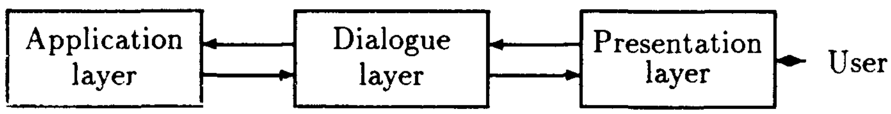

Figure 2.1: The Seeheim model of a UIMS.

In the Seeheim model there are two streams of data. The application layer generates output and sends them to the dialogue layer, which in turn sends data to the presentation layer, causing changes on the display. The other stream contains the input data, originating in the presentation layer as a result of user input, and flowing through the dialogue layer to be received by the application layer as input.

In all the dialogue notations based on the Seeheim model it is conventional to consider data to be quantized in tuples called *tokens* or *events*. We shall use both terms interchangeably, although sometimes an event is defined as conveying no information other than the type of event and who it came from.

2.3 The Application Layer

The *application layer* does whatever the "real work" is supposed to be. If the goal is to solve a set of linear equations, then the work may be inverting a matrix.

The entry of data and display of the results is not the responsibility of this layer. Informally, the application layer is what would remain in a program after all the portions of code dealing with input/output with the outside world were excised—hence the desire to move interface functions to other layers. This description is oversimplified because a graphical program is more complex than an interface stuck onto an application layer. Feedback is often crucial to the smooth use of a graphical program and often must make use of information available only in the application layer. Take for example a line art editor, which must be aware of objects stored in its database and alter the display appropriately when the "delete" function is selected. To the "real work" we must also add error checking and semantic feedback as some of the duties of the application layer.

2.4 The Presentation Layer

The function of the presentation layer is to insulate the program from physical device dependencies. The presentation communicates with physical devices that accept input or present output on the display and presents these as logical devices to the dialogue layer. An example of a logical device might be a valuator that returns a real number between 0.0 and 1.0. The presentation layer may use different realizations for this: sliders of differing shapes and sizes, knobs, perhaps even a pair of up/down buttons coupled with a digital meter. These may be alterable at compile time, perhaps even at run time. Another example of a physical/logical translation is the simulation of a middle button on a two button mouse by *chording* (depressing both buttons simultaneously).

Another concern of the presentation layer is the placement and sizing of graphical objects. The appearance of objects may change depending on the aspect ratio and size of the screen on which it is presented. A program should be able to work with many different hardware configurations without the need to recompile. The appearance of a graphical object may depend on the importance of the object at a given time in a session. For a window that has been iconified because it is not in use, a page of text might be represented as a rectangle with barely visible scribbles. When the user decides to work on that window and opens up the icon to zoom in on the page the presentation layer should show the page in greater detail.

Finally, the presentation layer allows users to customize an interface to individual preferences. A command may be entered in several ways. One user may be content to access program functions via menus. One user may prefer to use mouse buttons for speed. Such convenience bindings of buttons to program functions are commonly called *accelerators*.

Many existing windowing systems do not have a clearly distinguishable presentation layer. However some of the services provided by a presentation layer

are available in popular windowing systems. An example is the X Windowing System [Scheifler and Gettys, 1987], which defines a protocol between the window server and clients. The server manages the layout of the screen and multiplexes the input devices between the windows.

The need for a presentation layer is generally accepted. Singh [Singh, 1985] has shown how, for a particular UIMS, all device dependencies can be concentrated in the presentation layer, and how libraries of interaction techniques can be created to provide physical to logical mappings appropriate to each program. The presentation layer can enhance program portability. A common interface can be presented to a program by a set of libraries, each tailored to a different windowing environment.

It is useful to divide the presentation layer into two parts: the environment dependent part, which interfaces with hardware; and the environment independent part, which implements logical devices and handles user preferences.

Current windowing systems tend to blur the distinction between these two parts because libraries hide these functions from user programs. For instance, in the X windowing system the environment dependent portions are all in the server process, but some environment independent functions are also handled in the server. In the SunView[Sun Microsystems, 1986] windowing system, both the environment dependent and environment independent portions are implemented by library functions. This variety in implementation strategies is a clue that the boundary that surrounds presentation functions is not as clear-cut as the Seeheim model seems to imply. We shall return to this point in Chapter 5.

2.5 The Dialogue Layer

The dialogue layer is the seat of control in an interactive program. It determines the structure of a conversation by constraining communication to permissible patterns of interaction. The dialogue decides what constitutes a unit of information that should be passed (in either direction) between the user and the computer. It also classifies the data and sends it to the appropriate thread of conversation. In short, while the presentation layer determines the visual appearance of a GUI, it is the dialogue layer that determines the quality of the conversations between the user and the computer.

In most textual programs, the dialogue portion is determined by the placement of calls to I/O routines within the code. Most programs do not have sufficiently complex interaction structure to require separating this dialogue portion. Compilers and other grammar driven programs are notable exceptions. The reason is that these programs have to accept or reject input data according to formal specifications and these specifications are complex enough that machine-generated parsers represent a significant saving in coding effort.

Simple GUI dialogues use the same technique as textual programs for constructing dialogues: calls to I/O routines and use of traditional control-flow structures in an imperative language. This technique is impractical with complex dialogues. Some GUIs provide libraries that have a rudimentary means of specifying dialogue For instance, the X Toolkit[McCormack *et al.*, 1988] allows the programmer to specify translation tables from events to semantic actions. While such libraries are a form of programming automation, there are still deficiencies. The specification language often does not allow for anything more complex than a sequence of events to be specified and thus fails to be sufficiently powerful. The dialogue specification is tied to a particular window system and therefore non-portable. Only a formal dialogue notation can satisfy all the goals of a GUI specification system enumerated in Chapter 1. Our work with PENGUIN will show that a formal dialogue notation aids in the creation and maintenance of GUIs.

Several formal notations for dialogue have been proposed. A dialogue notation is intended to capture details of the sequence and structure of a "conversation" with the user. There are three major categories for notations currently in use: transition networks, event handlers, and grammars. There are also notations such as "input tools" which do not readily fit into these categories but partake of some features of each. We evaluate how well each notation supports the needs that derive from the characteristics of GUIs. We also evaluate each notation according the four desirable goals we proposed in chapter 1 for a GUI language.

2.5.1 Transition networks

Transition networks have been proposed by Newman [Newman, 1968], Edmonds [Edmonds, 1981], Jacob [Jacob, 1983] and Guest [Guest, 1982], among others.

Transition networks consist of nodes and arcs. Nodes correspond to states of the system, arcs to events that cause changes in state. Events may originate with the user or with the computer. Recursive transition networks (RTNs) are needed to provide power equivalent to that of context-free grammars (CFGs). Augmented transition networks are RTNs augmented with registers [Woods, 1980] for argument passing. Jacob uses a linearized textual notation for specifying transition networks, as does Guest, for ease of entry using textual input devices.

This notation is easy to learn because each state of the dialogue corresponds to a location in the network. Simple dialogues can quickly be constructed in this notation. Transition networks suffer from the drawback that transitions between arbitrary states are permitted. Without voluntary discipline in the construction of transition networks, a programmer can create a convoluted network that is hard to maintain. This problem is analogous to the unrestricted use of GO TO statements in programming. It is also easy for the programmer to make mistakes transforming a state diagram into a textual representation. There is no support for

multi-threaded interaction; the dialogue is in exactly one location in the network at any time. Exceptions are difficult to express. The most promising approach, wild card arcs for exceptions, can handle only one level of exceptions.

Transition networks satisfy the translation and portability goals, but fail the specification and reusability goals. Diagrams get complicated quickly. Using a textual translation of a diagrammatic language increases vulnerability to transcription errors. There is no multi-thread support. Exception support is poor. As there is no concept of sender identity, it is not possible to dispatch events to different instances of a sub-diagram. It is also not possible to reuse a sub-diagram by modifying it into a slightly different sub-diagram without renaming events (arcs).

2.5.2 Event handlers

Event handlers were originally proposed by Green [Green, 1985]. ALGAE [Flecchia and Bergeron, 1987], is one implementation of event handlers.

A handler is a piece of code with entry points. An entry point is activated when a suitable input event arrives. Handlers contain internal state in the form of static variables. Access to global state is also permitted. Events may be received from the presentation or from the application. Events may also be sent from other handlers by specifying the recipient in the send operation. The entire collection of handlers resembles a Smalltalk system in which objects activate other objects by sending messages. Handlers can be created dynamically, each instance sharing the same code template but having its own private variables. Event handlers can easily handle multi-threaded conversations and exceptions.

One drawback is that there is no syntax for specifying a sequence of events interleaved with actions. Each entry point consists of the activating event followed by the action code. This also implies that only one event can be processed at a time. In contrast, grammars allow sequences of events to be specified.

Another drawback of event handlers is that the number of handlers that may have to be activated in response to an event may potentially be unlimited. The notation cannot guarantee that no two handlers have event types in common. As every possible event handler must be examined, the implementation may create a performance bottleneck.

Event handlers satisfy the translation and portability goals, but are weak for specification and reusability purposes. The limitation of an entry point to a single event instead of a sequence of events means that some dialogues will require unnatural expressions in this model. (Appendix B shows an interface that makes good use of the concise notation provided by grammars, such as closure operators.) The lack of a means of labeling external events by sender limits reusability to the creation of multiple instances from a single template. It is difficult to recycle code from one handler without renaming the events to avoid alphabet conflicts.

Finally, the distinction between internal and external events is artificial and can be subsumed in a more general model.

2.5.3 Input tools

Van den Bos [van den Bos, 1988; van den Bos, 1979] proposed an augmented BNF notation for specifying what he called "input tools." His notation partakes of the flavor of both context-free grammars and event handlers.

Input tools are pieces of code that accept primitive inputs (like a keypress), possibly perform some computation, and return values to callers. More sophisticated virtual input devices are composed from basic input tools. For example, a tool can be specified that accepts only a decimal string. The major contribution of this work was to allow operators for concurrency. For instance, `a & b` in this notation means that both tools `a` and `b` must successfully execute for the expression to return a success status. This notation allows multi-threaded dialogues to be specified.

There has been at least one attempt to implement Van den Bos' proposal. A translator generated a parser from the specification language[Matthys, 1985]. The experiment was not regarded as successful for two reasons. The major problem was performance. The reaction time of the parser became unsatisfactory as the number of productions increased because locating the tool that would accept the input required exhaustive search of all active tools. The second problem was that the design notation was not easy to work with. Side effects occurred at unexpected times.[1] The poor performance only aggravated the problem by forcing programmers to seek the most economical, instead of the most natural, expression.

Input tools satisfy the translation, portability and reusability goals, but are less than satisfactory for specification. Each production in the grammar is enclosed in an input tool, which makes it difficult for a collection of related productions to share common state. As with event handlers, the notation may require sending events to an arbitrary number of recipients, an efficiency concern. As defined, there is no mechanism for specifying and sending directed events. Finally, the language does not provide persistent local storage.

[1]The authors do not elaborate on this criticism, but they apparently refer to the side effects caused by executable statements. The notation does not provide the means to define persistent local storage, so state information has to go into the global area. Any input tool that has been activated can change the shared global state.

2.5.4 Context-free grammars

Context-free grammars in dialogues reuse the large body of language theory applied to the construction of compilers. Proponents of context-free grammars include Olsen [Olsen, 1983; Olsen and Dempsey, 1983], and Hanau and Lenorovitz [Hanau and Lenorovitz, 1980].

Context-free grammars describe a dialogue by defining the language of all the legal sequences of user input to the application, or of application output to the display. The major advantage of CFGs is conciseness. However there is some evidence that this conciseness makes CFGs harder for programmers to learn and use[Guest, 1982].

Olsen uses a notation similar to LL(1) grammars. His Interactive Pushdown Automaton (IPDA) executes the algorithm corresponding to the compiled grammar. An IPDA is like a PDA, but has been augmented with three features: conditional transitions to allow semantic control of the parse syntax; immediate transitions to allow semantic actions without prior input, and pervasive transitions to handle exceptional conditions. The last feature allows the user to escape from the current interaction and resume the interrupted conversation later. The IPDA as described has not been generalized for multi-threaded conversations. As a result, only one level of escape is allowed. It is not possible, for example, to get further help within the help dialogue. This problem cannot be solved easily without providing a true multi-thread capability.

2.6 Project Goals and Context-free grammars

Context-free grammars have certain advantages over alternative specification methods. These advantages are:

- Grammars are concise. The representation of a dialogue using grammars is smaller than that using a transition network and about par with event handlers. The economy of symbols makes it easier to comprehend the dialogue as a whole and eases modification and maintenance.

- Grammars have structure. There is incentive for the designer to factor out common pieces of the dialogue. The semantics of the parsing algorithm can be designed to support composition of dialogue fragments to form larger dialogue sections. This encourages the reuse of existing dialogue code by programmers.

- Techniques for parsing grammars are well studied. The performance of a grammar based dialogue can be predicted well from the algorithm.

- A grammar concisely codifies what constitutes a group of legal input items and when they should be acted on.

A disadvantage of context-free grammars is that the notation is partly declarative, which can be disconcerting to programmers used to writing imperative code.

The success of tools for turning grammars into parsers lends support to the hypothesis that the translation goal is achievable. Portability and reusability are also important goals in other computer languages and the techniques for achieving these goals are well understood and can be applied in a grammar-based dialogue language.

2.7 Summary

Previous notations for dialogues are inadequate because they fail to address fully the peculiar needs that are a consequence of the characteristics of GUIs. Transition networks fail to handle token attributes or multi-threaded interaction. Event handlers support multi-threaded interaction but do not provide directed token dispatch, entailing considerable inefficiency. Input tools do not support directed token dispatch or private data. Traditional context-free grammar based notations do not provide true multi-thread capability. Of all the previous notations only event handlers have support for backing out of a thread. The design of PENGUIN, which the next chapter will present, remedies all the shortcomings of previous notations and fulfills the needs of GUI dialogues.

3 The PENGUIN Language

This chapter describes and motivates the features of PENGUIN. The grammar notation captures the sequencing of input. Fork productions specify multi-threaded behaviour. Modules define self-contained reactive objects that can be reused. The PENGUIN compiler provides the bridge from such high-level specifications into executable code. Implementation details are pushed into the run time library, enhancing portability of the application across different environments.

In common with other dialogue notations, PENGUIN's language model considers a *event* or *token* to be the basic unit of interaction with the outside world. Tokens are mapped into *terminals* in PENGUIN grammar productions. Within productions, time precedence relationships implied by the placement of terminals, non-terminals and actions are enforced. Grammars provide a concise way to specify these time precedences. The execution of a PENGUIN program is driven from below by the arrival of tokens and from above by the prediction of productions.

PENGUIN modules allow a program to be constructed as a set of smaller grammars instead of one monolithic grammar per program. Modules also provide data hiding and instantiation. Resources (variables) are private to each instance of a module. Reuse of module code is encouraged. PENGUIN *components*, described in Chapter 5, provide a mechanism for applications that need true concurrent execution. Components also allow applications to be built as co-operating processes, reducing the level of conceptual complexity of each component.

3.1 Fork Productions

Chapter 2 shows that an adequate notation for GUIs must allow the specification of interleaved threads of interaction so that the programmer does not have to obtain concurrency by other, less lucid means. Standard context-free grammars are insufficient in this respect. The foremost drawback is the strict order of evaluation in CFGs. CFGs receive input from one source and have one locus of control.

To remedy this shortcoming, context-free grammars[1] are augmented with two new types of *fork* productions: the parallel ***and*** production: **&>**, and the parallel *or* production: **|>**, in addition to the sequential *derives* production: **=>**. Both new types of production cause the creation of ***sub-parsers*** (Section 3.2) which work in parallel. Formally stated:

> A production L **&>** $R_1 R_2 \ldots R_k$ succeeds when all of the symbols on the right hand side succeed. More formally, the **&>** production generates all interleavings of strings generated by the symbols on the RHS.
>
> A production L **|>** $R_1 R_2 \ldots R_k$ succeeds when one of the symbols on the right hand side succeeds. More formally, the **|>** production generates all interleavings of a string generated by one RHS symbol with prefixes of strings generated by the other symbols on the RHS.

`&>` productions are useful when the order of input is immaterial. Filling in a form is one example:

```
form &> A B
```

Both `A` and `B` must appear in the input for `form` to succeed. The terminals that `A` and `B` derive may be interleaved in any order in the input.

`|>` productions are useful when the completion of one sub-production obviates the need for others. User-generated interrupts (e.g. from hitting the *DELETE* key) are one example:

```
getnumber |> interrupt read_digits
```

The completion of either `interrupt` or `read_digits` will cause `getnumber` to succeed. Different levels of interrupts can be used to back the parser out to arbitrary pre-arranged positions.

The **&>** and **|>** productions allow us to construct a hierarchy of sub-parsers to manage a conversation with multiple levels of aborts, nested parallel conversations (e.g. for interactive help), and other useful structure.[2] This feature of PENGUIN is

---

[1] We use a hopefully self-explanatory extended Backus-Naur Form notation.

[2] It is worth noting that our augmented grammars are not in general context free. The language generated by

```
A &> B C
B => a B c | ϵ
C => b C d | ϵ
```

is one example; its intersection with the regular set $a^\star b^\star c^\star d^\star$ is $a^n b^m c^n d^m$. Thanks to David Sher for this proof.

similar to the light-weight process or thread features of some high-level languages and programming environments.

A more detailed example of the use of fork productions is found in section 3.7.

There is one additional rule on the use of fork productions: it must always be possible to predict their use without lookahead. In other words, if A is the left hand side of a fork production and αA is a prefix of a valid sentential form, then *every* valid sentential form beginning with α must begin with αA, or be derivable from a sentential form that begins with αA. This amounts to insisting that concurrent conversations must be started explicitly by user or application action, and need not be detected in response to the arrival of input from one of the branches. This rule simplifies parsing considerably and is also consistent with natural dialogue structure.

Two other types of fork productions are provided. These are **&:** and **|:** productions. These "no-wait" versions are defined just like **&>** and **|>**, except that the parent continues to accept further tokens without waiting for the RHS items to complete. These productions are useful when there is no need for the children to trigger any actions after completion. The use of these productions simplifies certain common situations in dialogues.

3.2 Sub-parsers

Formally, a *sub-parser* is a thread of execution in the parser created by a fork production. Except in the no-wait versions, the parent thread is suspended until all child threads are terminated, whereupon execution resumes in the parent thread.

Although any number of sub-parsers can be created during parsing, there can only be as many distinct domains for sub-parsers as there are *sub-grammars* (Section 3.5).

3.3 Action routines

In the above examples, *actions* have been omitted for clarity. Actions are pieces of code that may be interposed anywhere in the right hand sides of productions. They are used to effect operations that change the state of the program or cause other side effects. Actions are triggered at moments in time consistent with their position in the productions activated. Actions may be considered items that always succeed but cause side effects by executing code. Some actions are routines that output to the presentation layer or to the application layer. In the Seeheim model tokens for the dialogue layer arrive from the other layers. In the generalized PENGUIN model discussed in chapter 5 the presentation is a special case of a

PENGUIN *component* that manages physical resources. Multiple dialogue and application components may exist. All communication between components is through tokens. The output of some component becomes the input of some other component.

3.4 Context

Fork productions make it possible for PENGUIN grammars to have more than one active production. This enhancement creates the problem of dispatching tokens. For example, if two windows created by a fork production are active, then there may be two sub-parsers with the same production active and we need to decide which sub-parser should receive an incoming token.

One might require every sub-parser to have a separate alphabet so that at most one production can accept an incoming token. However, the number of sub-parsers existing at run time is in principle unlimited, since those sub-parsers might correspond to different windows of the presentation component. If an unbounded number of windows can be created, it becomes impossible to enumerate all tokens in a finite grammar at dialogue-generation time.

We stipulate that all tokens must possess at least two synthetic attributes: *value* and *context*. Value captures the usual notion of token type. Context allows the dialogue to differentiate between tokens of the same value from different sources. As with other attributes, context may be assigned and tested in the actions of the grammar. In addition to matching in value, each token must also match in context. The context of an incoming token must match the context "inherited" by that token in the grammar. No copying is actually required for this inheritance; the compiler merely has to note the location in the attribute record of the symbol deriving the terminal where the matching context can be found.

A notation based on left-attributed LL(1) grammars in single-assignment form [Aho *et al.*, 1986; Knuth, 1968] manages the use of contexts. The inherited attributes of a symbol X in a production of an L-attributed grammar depend only on attributes of RHS symbols to the left of X or on inherited attributes of the LHS of the production. The interactive nature of the parser requires this; if the grammar is not left-attributed, more than one pass may be required to evaluate attributes and this conflicts with the interactive nature of the grammar. Attributes that contain context values are identified to the dialogue compiler. In an LL(1) grammar, all attributes can be evaluated left-to-right, in the course of the parse itself. The single-assignment property requires that all dependencies be copy rules; computation is performed solely in action routines. The restriction to one token of lookahead is consistent with intuitive behavior for interactive systems.

3.5 Sub-grammars

The fork productions previously defined partition a grammar into *sub-grammars.* The productions of a sub-grammar are those productions derivable from start symbols, without traversing fork productions. Context propagation obeys the same copy rules as other attributes. However, in the presence of fork productions, further rules must be imposed on context propagation in fork productions to ensure that two branches cannot both match a token. This is to preserve the property of PENGUIN grammars that no more than one sub-parser will respond to an input token, guaranteeing parsing efficiency at run time.

First, action routines that return context values in synthetic attributes are required to create new, unique values for each call. Second, the copy rules for a given production are not permitted to assign the same context value to two different inherited attributes of a non-terminal on the RHS or to two different synthetic attributes of the non-terminal on the LHS. Furthermore, no context value may be copied into both branches of a fork unless the value alphabets of the two sub-productions are disjoint. Simply put, the branches of a fork either (1) partition the token value alphabet between them, or (2) only know the names of different contexts.

A modified LL(1) parsing algorithm that handles these context matching and propagation rules was discovered. An alternative parsing algorithm requires us to impose the further restriction that sub-parsers inherit a fixed and statically determinable number of contexts. This means that synthesized contexts may not be used within the sub-grammar in which the action appears, but may only be passed to an inferior sub-grammar. With this rule it is possible to re-write each sub-grammar as a conventional context-free grammar, without context attributes or forks. Standard (e.g. LR) parsing algorithms can then be employed. The descriptions and analyses of both parsing algorithms are found in chapter 4.

Limiting sub-grammars to a fixed number of contexts is not as serious a restriction as it might at first appear. Artificial sub-grammars may be introduced to change to a new and different context:

```
S => ... X ...
X &> Y
```

The fork productions work as expected for the limiting case of one branch. In this example an `|>` production would have had the same effect.

3.6 Execution model

A PENGUIN program starts with the run time parser in control. The parser initially predicts the first start symbol of the grammar. While no fork productions

are encountered, parsing proceeds as in conventional context-free grammars: input tokens either match terminals in the grammar or predict productions. Actions interspersed between items are executed at appropriate times. Tokens can be sent to the outside world by routine calls in action code.

When a fork production is encountered sub-parsers are created. Each sub-parser accepts tokens independently of other sub-parsers; the language rules ensure that no more than one sub-parser will accept an input token.

In short, fork productions manage the hierarchy of sub-parsers and drive the program from the top down, while tokens arriving from the outside match terminals in the grammar and drive the program from the bottom up.

3.7 Examples

Here is a simple dialogue that awaits the key **k** and then creates two new sub-windows with an **&>** production. Both sub-windows must terminate before the parser proceeds. In the following, inheritance rules are written using a notation similar to argument passing in imperative languages: `Y(@X.c1)` means that the first attribute of `Y` is copied from the `c1` attribute of `X`.

```
terminal key(context ctx) = 'k';
nonterm S(context ctx), X(context c1, context c2);
nonterm Y(context c), Z(context c);
nonterm new(|context ctx1, context ctx2);
...
S => key(@S.ctx) new X(@new.ctx1,@new.ctx2) ...;
X &> Y(@X.c1) Z(@X.c2);
```

The non-terminal **new**, defined elsewhere, creates two new windows and passes their contexts to `Y` and `Z`, so that `Y` and `Z` operate in different contexts.

In practice much of this verbiage is eliminated by these default context rules:

1. Every symbol automatically has a context attribute called **ctx** declared. This attribute is placed after explicitly declared context attributes, if any. This means that a symbol with no context attribute declarations has one context attribute by default.

2. A RHS terminal or non-terminal `S` with $n-1$ attribute copy rules for n attributes is assumed to have the additional rule `S.ctx = L.ctx`, where `L` is the LHS non-terminal. Recall that by the first rule every symbol has at least one attribute. Thus the most common situation where the LHS

context is propagated to all RHS symbols can be obtained by declaring neither context attributes nor copy rules.

This next example illustrates the use of alphabet splits. The same context is used by both parts of the fork, but they respond to different keys.

```
terminal left = 'l', middle = 'm', right = 'r';
nonterm L, R;
...
S |> N A;
N => L+;
N => R+;
L => left ...;
R => right ...;
A => middle;
```

This grammar repeatedly chooses between two productions depending on whether the left or the right button is clicked, but finishes when the middle button is clicked.

3.8 Attributes

Besides the value and context attributes, tokens may carry other attributes. Tokens are the units of information in the PENGUIN model. For example, a VALUATOR (slider or knob) may inherit the attribute *range* to specify the range of real numbers it operates on and synthesize the attribute *value*. Actions embedded in the dialogue specification may manipulate these attributes.

In the PENGUIN model there is no restriction on the size or number of attributes that may be carried by a token, although a practical implementation may restrict the size of a token to the amount of data that can be read without fragmentation. Different graphics programs will have different notions of what constitutes a token. A drafting program may treat one keystroke as a token, while an image display program may consider an entire image to be a single token.

3.9 * and + Operators

In BNF the idiom for reading zero or more occurrences of terminal t is:

```
A => T
T => t T
T =>
```

This idiom is common enough to warrant shorthand. We can write this as:

```
A => t*
```

The + operator plays an analogous role for one or more occurrences of `t`. Nonterminals may also use these suffix operators.

In the current language specification, closure operators may not appear in the RHS of fork productions. An auxiliary symbol is required if a fork production derives a suffixed item.

Attribute flow rules for a suffixed item X are as follows: Inherited attributes, if any, of the X mean that every occurrence of the X receives those attributes. The last occurrence of X produces the synthesized attributes specified for X. If no instances of X are accepted, then the synthesized attributes are left unchanged.

Besides allowing the programmer to write concisely what is intended, these operators also allow the compiler to optimize the grammar to prevent semantic stack buildup. The copy actions for attributes are also simplified.

3.10 Error Recovery

Error tokens are tokens that cannot advance the state of the parser. There may already be explicit productions in the grammar to handle some user errors. It is unacceptable to drop error tokens gratuitously. At least the default action should be to reflect the error to the presentation component so that the user is aware of the error. There are two ways a token is unacceptable to the parser.

1. The context of the token is acceptable to some sub-parser but the value of the token is not legal at this point in the parse.

2. The context of the token is not acceptable to the parser because there is no active sub-parser that will accept this context.

The first case is the same problem as a syntax error in conventional parsing without context. Aborting the parse on such an error is unacceptable. An interactive parser should never terminate on detecting an error, unlike a batch parser. For conventional parsers, generally some form of error repair and recovery may be attempted on the program. For interactive applications, error repair is not as

important. The error is usually detected at the first offending token and is usually the result of a typing error by the user.

The keyword `catchall` denotes a terminal in the grammar that is allowed to match tokens not acceptable to any other production. The `catchall` token is subject to the same context binding rules as ordinary tokens. This allows a hierarchy of `catchall` tokens for recovery at different levels of the dialogue. In the absence of an explicit `catchall` thread in the grammar for a given context, a default error handler is installed. The default error handler is likely to do something simple, such as sending an out-of-band signal like a bell character to the presentation component.

There may be several causes for the second situation. The thread may not exist yet because the input has arrived before the creation of the matching context. This usually means the programmer has failed to make sure an sub-parser is active whenever a token with its context could be generated, e.g. allowing user input in a window before the corresponding sub-parser has been started. The thread may not exist because all threads for a given context have exited. This may happen when input is buffered ahead and the sender continues to provide tokens after the last acceptable token.

Neither case admits to run-time repair. In the first case we have a programming error. In the second case, the extra tokens are superfluous. Both contingencies are dealt with by a top-level error handler that prints a diagnostic message about stray tokens. The programmer can provide an error handler to replace the default one supplied by the runtime library.

3.11 Modules

A graphics program that contains a single, monolithic grammar is not only hard to write and maintain, but also discourages the reuse of grammar fragments. One of the goals of PENGUIN is to promote the reuse of software.

Dividing a dialogue into *modules* eases construction and maintenance. Several general criteria apply when dividing a dialogue into modules. Each module should have a clean protocol, specified by a grammar, for its interaction with other modules; the separation should be made at a point that is most likely to allow the module to be reused later; the module should be easy to parameterize; and any internal state should be kept local to a module. The criteria stated resemble those recommended for writing data abstraction routines, with the chief addition of grammars for handling protocol.

Modules can be separately compiled and linked later, a small efficiency gain for programs comprising large numbers of modules. Libraries of modules will encourage the reuse of software. Unnecessary detail can be hidden from programmers.

All this is similar to object code modules in programming environments. Another saving results because the combined space requirements of a composed module will be less that of a single monolithic module because the number of contexts in the sub-parsers will be less than the total number of contexts in the whole parser.

Formally a module contains

- A grammar to specify its interaction protocol with the outside.
- Action routines, for computation and sending output tokens.
- Local variables for retaining any state of the module between activations.

Information enters the module in four ways: contained in input tokens, inherited through attributes of start symbols, or through environment key-value pairs.

For GUIs, a module will generally correspond to an interaction object. An example is a graphical slider. Its interface has input attributes, namely, the size, location, and orientation; and yields an output value, the setting of the slider. The device maintains internal state, the current set value. Some modules have no resources to manage but merely handle protocol.

The text of a PENGUIN module defines the *template* of a module. An *instance* of a module is created with an explicit invocation of the **create** symbol of that module and a module handle created for grammar predictions involving that instance. Thereafter, the module handle is used to qualify references to other symbols of the module. The **destroy** symbol invalidates the handle and makes it available for another create operation. All instances of a module shares the parse tables of the template, but are otherwise separate in their lives.

Here is an example showing the life-cycle of a module instance.

```
module M mh;
...
S -> mh:create mh:S mh:destroy
```

The **create** action assigns the handle to **mh**. Next the start symbol S of this instance is predicted. This prediction may in turn cause subordinate module instances to be created. When this start symbol has completed, the instance is destroyed and the handle mh is invalidated.

In practice, the creation, use and destruction of module instances are usually in separate productions because they are used in different phases of a program's lifetime.

3.12 Environment variables

Modules often need to obtain information from the outside that may differ between invocations. Attributes may be used to pass this information but are cumbersome if the information remains unchanged across most calls.

Environment variables address this need. The environment is a set of (*name, value*) pairs. *Name* is a legal environment name in PENGUIN, and *value* is an uninterpreted block of bytes. Although it is conventional in some programming environments to restrict value to the domain of printable strings, PENGUIN does not require this.

A given set of environment bindings is inherited by all descendant modules. Changes in the current binding do not affect bindings already passed to descendants, nor can descendant modules affect bindings in superior modules. PENGUIN provides a syntax to obtain the binding for an environment name. Routines in the run time library allow new bindings to be created or old bindings to be superseded.

This environment mechanism handles many of the name to value translation needs of a graphical environment. One example is obtaining the user preference for the background colour of a window, which usually remains unchanged across many module instances.

Here is the previous example enhanced with a use of an environment variable.

```
module M mh;
...
S -> mh:create mh:S($colour) mh:destroy
```

`$colour` is an environment variable that has been inherited through the hierarchy of module instances. Here it is passed as an attribute to a start symbol of module `M`.

3.13 Queues

Although sub-parsers work in parallel, the granularity of this parallelism is at the token level. Since there is never any ambiguity as to which sub-parser will accept a token, one input queue suffices for a PENGUIN parser no matter how many sub-parsers are active. The queue is ordered by arrival time of the tokens regardless of the source of the token.

Another way of visualizing the situation is to say that there is only one parser in a PENGUIN component and that this parser multiplexes tokens from the input queue between its many active sub-parser threads.

In theory, one input queue suffices, but in practice, there are tokens that must arrive ahead of other tokens, such as user interrupts and other out-of-band data. A second queue, called the urgent queue, is needed. Tokens from this queue are always read before tokens from the normal queue. This in effect provides a two-level priority scheme for tokens. An arbitrary amount of data may be waiting in the normal queue. If out-of-band tokens have to wait for all preceding tokens to be processed, it defeats the purpose of the out-of-band information. Note that nothing about the speed or latency of the connection between components has been assumed. The PENGUIN model accommodates both fast, local connections and long, high-latency connections.

4 Compiler and Parsing Algorithms

In this chapter we look in detail at the algorithms and data structures of PENGUIN.

There are two kinds of algorithms to be considered—those in the parser generation phase and those in the run time phase. Although the former precedes the latter chronologically, we discuss the latter first because the run time parse algorithm motivates the parser construction algorithm. We are interested in the run time space requirements of the parser, so we also analyze the data structures. The run time complexity and data structures of the parse generation phase are of little interest here but it is assumed a reasonably efficient implementation is chosen so that compilation will be fast enough.

4.1 Cactus Stack

In PENGUIN programs there is no longer a single focus of input in the parser, so a simple stack will not suffice. Since branches may be destroyed in any order we need a cactus parse stack.

One translation scheme for PENGUIN code is to generate data structures for an interpretive parser. Here the cactus stack is maintained by the parser and action code is called via procedure pointers. Another translation scheme is to realize the grammar as a multithreaded recursive descent parser. Productions and symbols become coroutines. The cactus stack is embedded in the stack frames of the threads. Action code is inserted inline in the routines corresponding to the productions.

In the interpretive method the cactus stack is a tree of production frames with two-way links. We need the bidirectional links because when a child terminates and its production needs to be exited, the sibling branches have to be located. Each frame corresponds to an instance of a production waiting to be completed. When a fork production is encountered an appropriate frame is constructed and the stack is split. The implementation of **&>** and **|>** productions is simple: a

counter in the parent tracks the number of child branches still unsatisfied. Every time a child branch is satisfied, the count is decremented. When the count reaches zero, the fork is satisfied and any remaining children can be aborted. For **&>** forks the count starts at the number of children and for **|>** forks it starts at 1. Figure 4.1 illustrates a branch in the cactus stack.

In the coroutine method when a fork is encountered the run time code creates a new thread. This involves allocating a chunk of memory to each branch for its stack. At creation each stack is initialized at the bottom with a return address that points to a fork cleanup routine. When a branch terminates, this cleanup routine acquires control, preventing execution from resuming at an indeterminate address. The cleanup routine is also responsible for dealing with sibling branches. If the backlinks implied by the return addresses of the threads are traced, the stack frames form a cactus stack structure. The bookkeeping code for **&>** and **|>** forks follows similar lines.

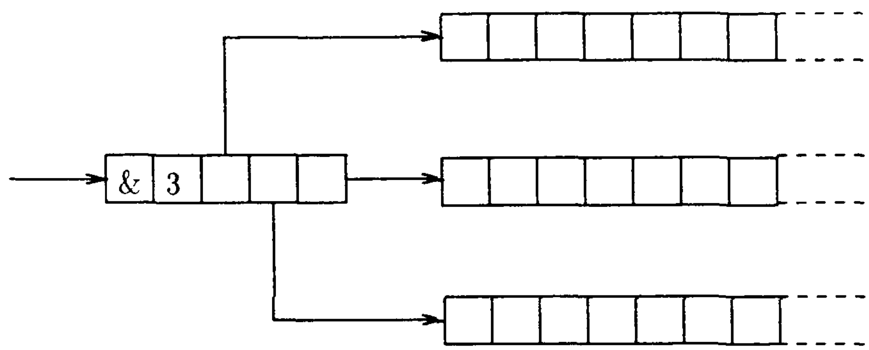

Figure 4.1: Fork junction in cactus stack

4.2 Parsing

The parser is the heart of a PENGUIN program. It realizes the fork productions, parses using context and propagates attributes.

Two approaches to parsing the input stream in the presence of context are presented here: use the context to dispatch tokens at run time (LSEARCH), or pre-translate tokens with context and value attributes to tokens with just the value attribute (TRANS).

The analysis presented here is per sub-parser. The worst case time complexity of a PENGUIN parser is dominated by the time complexity of the slowest sub-parser. The space requirement of the parser is the sum of the space requirements of the sub-parsers. Therefore, unless otherwise stated, all measures of time and space are per sub-parser.

Conventions used for symbols:

$\mathcal{P}$ Set of sub-parsers

i Length of input string

T Set of terminals in sub-parser

t Number of terminals (size of alphabet) in sub-parser

N Set of non-terminals in sub-parser grammar

n Number of non-terminals in sub-parser grammar

C Set of contexts of sub-parser

c Number of contexts of sub-parser

p Number of productions in sub-parser

In PENGUIN some non-terminals are non-predicting. That is, they expand immediately without lookahead of terminals. The compiler requires that the left hand side of any fork production be a non-predicting symbol. Also, non-terminals that appear on the left hand side of only one normal production are declared non-predicting by the compiler. In any case, a non-predicting non-terminal must appear exactly once on the left hand side in the grammar. For the purposes of parse table generation, the non-predicting symbols and associated productions effectively drop out of the grammar.

4.3 Locating sub-parser by context

When a token arrives, first its context attribute is used to determine which sub-parser could accept this token. This portion of the algorithm is common to both LSEARCH and TRANS. There may be more than one sub-parser that could match the context but not more than one will accept; this arises from alphabet splitting. A dictionary must be provided to map contexts to sub-parsers. Formally,

$$H : C \rightarrow \mathcal{P} \cup \{\epsilon\}$$

One realization of H is a hash table. It has to be updated whenever the set C changes. Under the restriction that each sub-parser inherit a fixed and statically determinable number of contexts, C has to be updated only at forks. Otherwise H must also be updated in response to actions that introduce new contexts.

The result of this mapping operation is either a pointer to one sub-parser or (in the case of LSEARCH) a pointer to a table giving the sub-parser for each token value for the case where alphabet splitting is used to dispatch to different sub-parsers. Figure 4.2 depicts the mapping operation.

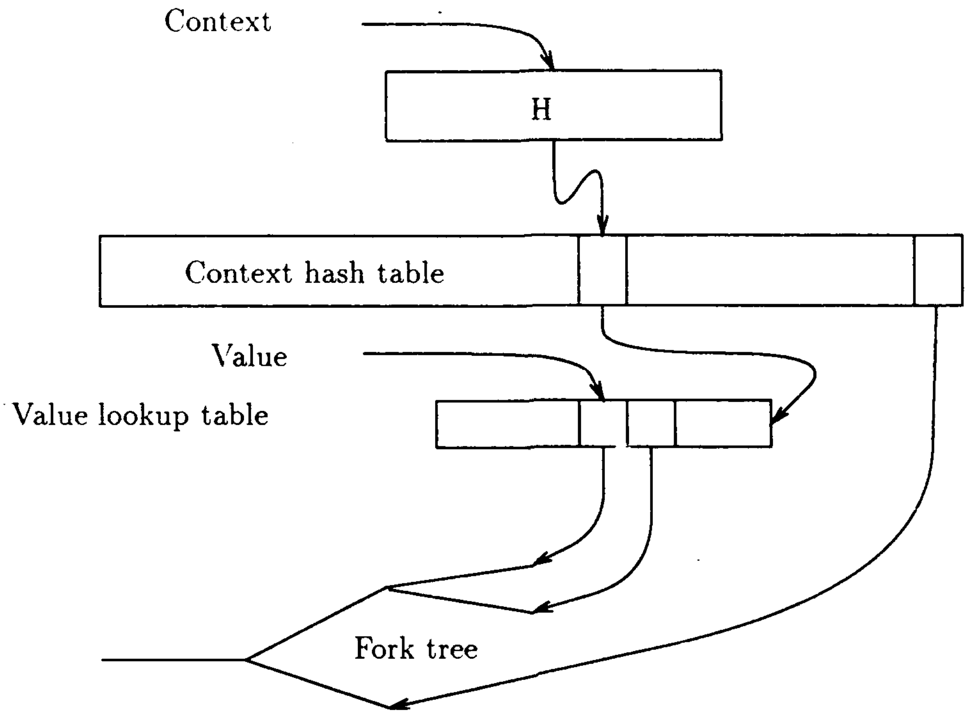

Figure 4.2: Data structure for sub-parser dictionary

4.4 Linear search for production (LSEARCH)

The algorithm described below is for the interpretive method. The sequence of events in the coroutine method is similar except that there is no external parser. The steps of the algorithm are executed by the code that the compiler generates or by the run time library routines.

After the sub-parser has been determined, the token value is used to advance the state of this sub-parser, if possible. In the presence of multiple contexts within a sub-parser, LSEARCH does a linear search through the list of valid contexts for this sub-parser to select a production based on context and value.

If the token is not acceptable to the sub-parser then error handling is invoked. This method is only usable with predictive parsing because bottom-up parsing would involve matching contexts against attributes that have not yet been inherited because they are in left corners, so the correct reduction has not yet been determined.

With the sub-parser in hand, we can now attempt a match. The case of a terminal on the top of stack is trivial. In the case where the input token is in PREDICT(TOS), we obtain a list of possible productions to predict.

parse:

```
push starting symbol of grammar on stack
loop
    while top symbol of some branch is non-predicting
        if symbol heads normal production
            push production on stack
        else if symbol heads fork production
            create new sub-parsers for each branch of fork
    let a be the input token and b its context
    determine unique branch that might accept token a with context b
    let x be top of stack symbol on that branch
    if x = a and context of x = b
        consume input symbol
        pop stack
        continue loop
    if x ∈ N
        let L be list of pairs pointed to by M[x,a]
        /* try productions that predict a */
        foreach pair on list with known context
            if b = context at field offset
                pop stack
                push production on stack
                continue main loop
    /* now try predicting epsilon and try to match TOS again */
    if ϵ ∈ FIRST(x)
        pop stack
    else
        error
/* end loop */
```

$M[X, a]$ is the production table indexed by non-terminal X on the top of stack and current input terminal a. Each element is a list of pairs of (field offset, production). The field offset is the offset in the TOS attribute record at which the matching context is found, and is precomputed in the table generation phase. Some entries on the list may have unknown field offset. It turns out that the RHS of the corresponding production will always be ϵ, or something that derives ϵ. These arise from productions predicted by terminals in the FOLLOW set of a non-terminal. In this situation there are one or more non-terminals on the parse stack that derive ϵ, that are above the non-terminal that derives the current input

terminal a. The context is unknown because we do not know how far down the stack the non-terminal bearing the matching context is. Fortunately, in such cases we can always predict the ϵ productions and remove non-terminals from the stack until the one bearing the matching context is exposed. Now either this sub-parser will match the context, or no sub-parser will.

4.4.1 Analysis

Mapping from context to sub-parser takes $O(1)$ time. Matching a terminal and pushing a production on the stack takes $O(1)$ time. When the top of stack is a non-terminal a list of up to c context to production pairs may have to be searched. Finding the pair with the matching context from this list takes $O(c)$ time. The worst case time complexity to parse an input of length i is then $O(ci)$.

We need $O(c)$ space for a hash table to dispatch by context and $O(ct)$ space for the lookup tables for split alphabets in the worst case.

The maximum amount of space required for the parse table is c times the original parse table because M is a table of lists each of maximum length c.

4.4.2 Table construction algorithm

If the top of stack non-terminal does not predict ϵ on the current input token, the context that must be matched by the input token is always in the symbol on the top of the stack. There are two cases: (1) The top of stack symbol is a terminal and matching context has already been copied into its frame. (2) The top of stack symbol is a non-terminal and a chain of copy rules exists from one of its attributes to the terminal that will eventually match the input symbol.

The fieldnames of the matching context in the top of stack non-terminal are precomputed with a straightforward modification of the FIRST set construction algorithm. FOLLOW sets will be treated later. In this modified algorithm, a FIRST set contains not just a set of terminals but rather a set of triples of (terminal, fieldname, production) where $(a, o, r) \in \text{FIRST}(X)$ iff $r = X \Rightarrow B\alpha$ where $B \stackrel{*}{\Rightarrow} a\beta$. Informally, r is the first production in the derivation of a from X and o is the fieldname in X of the context to be inherited by a. If X is a terminal, the fieldname is simply the fieldname of the (only) context attribute. This fieldname is readily computed from the inheritance rules specified along with the grammar. In the algorithm below, fieldnames for terminals in the FIRST set of a non-terminal are indicated by a prime mark.

fill_first_sets:

```
foreach non-terminal X
    FIRST(X) := {}
foreach terminal a
    FIRST(a) := { (a, o, ?) }
    foreach non-terminal X
        if ∃ a production r = X ⇒ a ... then
            let o' be fieldname where copy rule a.context = X.fieldname exists
            add (a, o', r) to FIRST(X)
repeat
    foreach production r
        add compute_first(r) to FIRST(lhs(r))
until no FIRST sets have changed
```

compute_first(r : production):

```
if length(rhs(r)) = 0
    return {ε}
result := {}
foreach symbol a in rhs(r)
    s := FIRST(a) /* current approximation */
    foreach triple u in s
        translate_fieldname(lhs(r),u)
        u.production := r
    add s to result
    if not ε ∈ FIRST(a)
        break
if ε ∈ FIRST of every symbol in rhs(r)
    add ε to result
else
    remove ε from result
return result
```

```
translate_fieldname(l : symbol; u : triple):

if an attribute copy rule u.fieldname = l.fieldname exists
    replace u.fieldname with l.fieldname
else
    grammar specification error
```

It is worth noting that, just as in conventional FIRST set construction, a terminal a can be added to FIRST(X) for non-terminal X multiple times.

FOLLOW sets are computed in the conventional way. There is no need to track the fieldnames because the fieldname is unknown for FOLLOW sets. Productions predicted by terminals in FOLLOW sets are of the form $X \Rightarrow \epsilon$ and non-terminals have to be popped off during parsing to discover the right context.

The FIRST and FOLLOW sets are then combined to form PREDICT sets.

predict:

foreach production $r = X \Rightarrow \gamma$
    add FIRST(γ) to predict_set(r)
    if $\gamma \overset{*}{\Rightarrow} \epsilon$
        remove $\{\epsilon\}$ from predict_set(r)
        add FOLLOW(X) to predict_set(r)

With the PREDICT sets, the parse table can be filled. Each entry in the table is not single-valued, as in a standard LL(1) table, but a list of (offset, production) pairs. The offset is that of the fieldname of the context attribute in the top of stack non-terminal at the time the prediction is made.

find_offsets:

```
foreach X, a
    M[X,a] := {}
foreach production r = X ⇒ γ
    foreach triple (a, fieldname, r) ∈ predict_set(r)
        add (offsetof(X,fieldname), r) to M[X,a]
foreach X, a
    if M[X,a] := {}
        M[X,a] := error
```

4.5 Translation to a new alphabet (TRANS)

TRANS has the advantage of being usable with any parsing method, and is not restricted to predictive parsing, provided the grammar is of a suitable form for the method in question.

The TRANS method converts a context augmented sub-grammar into a standard grammar. After the sub-parser lookup shown in section 4.3, the standard parsing algorithm is used to advance the state of the sub-parser.

At parser generation time, the provided grammar $G = (T, N, C, P, S)$ with context and value attributes is transformed into an equivalent grammar $G' = (T', N', P', S')$ with only value attributes. At run time an incoming token with context and value attributes is converted to a token with only a value attribute and presented to a standard parser.

4.5.1 Table construction algorithm

In the following discussion the grammar G refers to the set of productions comprising a sub-parser. Recall our stipulation that a sub-parser inherit a fixed and determinable number of contexts. This allows us to partition the productions of a PENGUIN module into self-contained sub-grammars, each invoked via a collection of start symbols. Each of these sub-grammars can be transformed independently.

We do not have to know the values of the contexts, only that they are distinct contexts. Let T' be the set $C \times T$. Each terminal in T in the augmented grammar is replaced by a symbol in T'.

Each non-terminal in N has a set of contexts associated with it. For every distinct permutation of contexts associated with a non-terminal replace that non-terminal with a new non-terminal from N'.

Finally, every production in P that was headed by a non-terminal in N must be replaced by several copies, one for each distinct variation in N'.

Assign ordinals to the context attributes of the start symbol S. Every non-terminal in the grammar derivable from S has some permutation of a subset of S's context attributes. Every terminal in the grammar derivable from S has one of S's context attributes. We wish to find N', the set of non-terminals with distinct permutations of contexts, and T', the set of terminals with distinct contexts. Let k be the index of the last context attribute. These two predicates hold:

$$S(1, 2, \ldots, k) \in N'$$

$$\text{if } N_1 \in N' \text{ and } N_1 \overset{*}{\Rightarrow} \alpha N_2 \beta \text{ then } N_2 \in N'$$

We generate these sets by a top-down search. Assume two sets, initially empty, for terminals and non-terminals. Each element of these sets is a triple comprising the terminal or non-terminal identifier, the permutation of context attributes, and a field used to hold the transformed name in the last phase of the algorithm. We assign ordinals to the context attributes of start symbol S. (X, {1,2,3}) is a distinct permutation from (X, {2,1,3}).

gen_sets:

new_terminals := {}
new_nonterms := {}
add_to_nonterms(S, initial_permutation)
rename_symbols

```
add_to_nonterms(s, i):

if (s, i, ?) in new_nonterms
    return
else
    add (s, i, ?) to new_nonterms
    foreach production headed by s
        foreach non-terminal X and permutation m in RHS
            add_to_nonterms(X, m)
        foreach terminal a and context y in RHS
            add (x, y, ?) to new_terminals
```

When all the reachable permutations of contexts have been generated, new terminal and non-terminal names are assigned to the third field of all triples.

For every production $r = X \Rightarrow \gamma$ in P, one or more productions $r' = X' \Rightarrow \gamma'$ are placed in P', one for every triple with first field X, and where X' is the third field of the triple. Every string of symbols γ is replaced by the string of symbols γ', where each name and context permutation is translated to a new name.

```
rename_symbols:

fill the third field of triples with unique names
foreach production r = X ⇒ γ
    foreach triple of the form (X, m, Y)
        add production Y ⇒ γ to G'
foreach terminal a with context y in G'
    replace a with b where (a, y, b) ∈ new_terminals
foreach non-terminal X with permutation m in G'
    replace X with Y where (X, m, Y) ∈ new_nonterms
```

4.5.2 Analysis

For equal basis of comparison, we measure a transformed LL(1) parser.

The time complexity of this parse algorithm is the same as that of the underlying parse algorithm, since the translation step only adds a constant amount of time per symbol. In the case of a transformed LL(1) parser, this is linear in the size of input.

The worst case space blow-up is computed as follows: The number of terminals increases from t to ct. The number of non-terminals increases from n to $c!n$. This is because there are up to $c!$ possible permutations of the attribute path for the context attributes carried by a non-terminal. The number of productions increases from p to $c!p$. Each production is headed by a non-terminal and we need to make up to $c!$ copies of each production.

The size of a LL(1) parse table is the sum of the sizes of the prediction table, which is of size nt, and of the production list, which is of size pl, where l is the maximum length of any production. After transformation, these sizes become $c!nct$ and $c!pl$, roughly a factor of $c!$ increase. We should also add the size of the translation table, which is ct, but this is swamped by the other two increases.

We also need $O(c)$ space for a hash table to select a sub-parser initially, just as in LSEARCH.

In practice the number of contexts in a sub-parser (not the entire grammar for a module) is likely to be small. Recall that a context represents a source of input tokens. A sub-parser will probably deal with at most two or three contexts. Thus the increase in storage requirements may be tolerable.

4.6 Conclusion

LSEARCH requires $O(ci)$ running time as compared to $O(i)$ for LL. LSEARCH requires $O(c + ct)$ additional space over LL for the value lookup tables and up to c times the space for LL parse tables.

TRANS requires $O(i)$ running time which is the same as LL or LR. The worst case space requirement is a maximum of $cc!nt + c!pl + c$.

LSEARCH has the advantage of requiring little extra space with a factor of c slowdown while TRANS has the advantages of a constant translation time overhead, and being usable with any parsing method. For small c, TRANS has tolerable space requirement blow-up. However, small c also tends to favour LSEARCH.

Since the number of contexts in a sub-parser c is normally small (say ≤ 3), LSEARCH will suffice for our parsing algorithm. The implementation is described in detail in chapter 6.

5 PENGUIN in the large

The previous two chapters described the methodology for the construction of single PENGUIN programs. A programming environment requires many programs working in concert, and whose interrelations usually cannot be preplanned. For the construction of large applications, we need additional mechanisms for composing software.

Too many current graphics programs are written to run as standalone processes. Often the only interprocess connection is with the windowing system. This prevents users from combining graphical programs in useful ways. This is partly due to the lack of standard representations for graphics data; textual data is easily organized as streams of characters or lines.

PENGUIN remedies these deficiencies with a model of intercomponent data transfer based on generalized tokens. The PENGUIN model takes advantage of the multi-threaded grammars of the language. We exhibit the PENGUIN model for the composition of interconnected programs and show how this model subsumes the older Seeheim model of GUIs.

The PENGUIN model also solves the problem of writing portable GUIs for programs that have a significant computing component or deal with unpredictable delays. Current solutions require the use of system specific methods of integrating multi-processing with the interface to the presentation component. In PENGUIN, all interactions are handled uniformly as tokens.

5.1 Previous work

The UNIX[1] [Ritchie and Thompson, 1974] facilities for interprocess communication has been widely emulated. The extension from linear pipelines to arbitrary networks of processes that has been proposed for textual interfaces [McDonald and Dix, 1988] is natural for graphics environments, yet few general results have been

[1] UNIX is a trademark of AT&T Bell Laboratories

realized. Two previous efforts at introducing networks of connected components for GUI programming are Fabrik and ConMan.

5.1.1 Fabrik

Fabrik [Ingalls *et al.*, 1988] is visual programming environment in which visual and computational user interface components can be "wired" together to build new components and useful applications. The vocabulary of Fabrik consists of objects that either interact with the user or perform some computation. One characteristic of Fabrik is the ease with which composite components can be tested, as the tableau is always "alive" during assembly. The internal dataflow semantics of composite components are handled behind the scenes.

Fabrik incorporates a type checking system to validate connections. Types supported include primitive types, bundled (record) types, array types and enumerated types. The definitions of compatibility are similar to those in a conventional language. Each primitive component assigns a type to each of its "pins" restricting input and output types. An interesting type is the unspecified type which has the ability to propagate types to other pins once a connection is made to a typed pin. For example, a Selector component has one pin of Number type which selects one of several input pins all of unspecified type to be propagated to the output.

The components of Fabrik are not free-standing programs but more akin to subroutines from a library which have been assembled with the help of the Fabrik interface builder. A compilation step creates a composite component that retains all the semantics implied by the network of components. This composite component can then be entered into the library for reuse.

Fabrik demonstrates the practicality of composing behaviour by composing interactive objects, and the basic support necessary for type checking connections between such objects.

5.1.2 ConMan

ConMan[Haeberli, 1988], for Connection Manager, is a high-level visual language that allows users to dynamically build and modify graphics applications. ConMan provides the user with a visual editor with which the user can connect the output of a component (program) to the next one. Each component interacts with the user via an *interaction frame* (window). The major difference between ConMan and linear pipes in UNIX is that ConMan supports arbitrary graphs. Changes to one component propagate to downstream components. Useful components in ConMan include a watch component for inspecting data, a tape recorder, and a mixer.

ConMan itself is a user process that runs under the window manager. When a component starts up, it notifies ConMan of the input and output ports it will use. After connections with peer components have been established, components use interprocess communication messages to exchange data. Data is exchanged as typed, variable sized textual messages.

ConMan demonstrates the worth of two goals: encouraging developers to break monolithic applications into functional components that communicate with each other using high level data structures, and allowing the user to design and extend applications.

5.2 PENGUIN Components and Ports

PENGUIN extends the idea of communicating components. The model is not restricted to specialized environments such as those defined by Fabrik or ConMan. Fabrik components cannot be used outside of the Fabrik workbench for free standing programs; Fabrik is a world unto itself. ConMan relies on all participating components adhering to a special protocol for interprocess communication. Programs that use different protocols cannot work with ConMan protocols. Instead of requiring components to use a specific protocol, PENGUIN only lays down general principles for intercomponent connections. The *behaviour* of a network of interconnected components is formally specified by the grammars of the individual components. Compatibility between components is also formally defined. The ConMan model of components is subsumed by the PENGUIN model. Moreover, network windowing systems in GUIs are treated as just another PENGUIN component, rather than as a special kind of program. The generality of the PENGUIN model comes at a price; programmers must design the interfaces between components; there are no ready-made interfaces. The PENGUIN compiler is the foundation of a suite of tools that can help the programmer construct intercomponent interfaces.

We define a *component* as a parser with a collection of modules that com prise an independent executable program. A component is the smallest unit of independent execution and corresponds to a process in many operating systems. The behaviour of a component is specified by the union of the grammars in the composing modules. The input alphabet is specified by the union of the input alphabets of the composing modules. Together, the grammar and the alphabet specify the *protocol* adhered to by the component.

A component communicates with other components via *ports*. A port is a rendezvous site for a recipient, the set of modules inside a component and senders, the peer components. To the programmer, a port is a named resource to which tokens can be directed. This extra level of indirection hides the identities of the

communicants from each other, making it easy to change comr. unicants, even on-the-fly if desired.

Components may be interconnected at run time to construct different configurations, to take advantage of the synergy induced by communicating processes, in the same way that UNIX pipes can be used to build composite programs. Collections of PENGUIN components are not restricted to being coupled in linear configurations but can be interconnected as an arbitrary network.

Note that the connections between components are deliberately left underspecified. They are only required to be have the FIFO property, be flow-controlled and be reliable. No constraints are placed on the mechanism, transfer rate or latency. Implementations may use shared queues, RPC, sockets, streams, virtual circuits or whatever is appropriate.

5.3 Matchmaking

Connecting components requires a matchmaking process. We need to know if two components are *compatible* before connecting them together.

Alphabet compatibility is defined as: Every terminal generated by the sender is in the terminal set accepted by the recipient. Two conditions must be satisfied:

1. The alphabet of the sender is a subset of the alphabet of the recipient.

2. Corresponding pairs of terminals in the subset are type compatible.

The first condition can be verified by comparing terminal sets. The second condition simply requires verifying type compatibility between corresponding terminal tuples.

We assume that it is possible to transparently transfer terminal tuples between heterogeneous machines, using encoding mechanisms such as Sun's XDR [Microsystems, 1986] or ISO ASN.1 [ISO/IEC, 1987] to handle byte ordering and padding differences.

We note in passing that that all UNIX processes using byte-streams are trivially alphabet compatible because (1) the common alphabet is the 8 bit code and (2) the only field of terminal tuples is a single byte. Another example is provided by processes that adhere to the "lines of text" interchange standard. Here (1) the common alphabet contains one terminal—the line, and (2) the only field in the terminal tuple is the data in the line. Again, all processes adhering to the standard are compatible, ignoring maximum line length restrictions.

Two further levels of compatibility can be defined, but no effective procedure exists for enforcing them. *Language compatibility* is defined as: A sender is compatible with a recipient if every sequence of tokens generated by the sender is

accepted by the recipient. *Semantic compatibility* is defined as: Actions triggered by the receipt of tokens are those intended. Semantic compatibility is usually defined operationally by the user.

Alphabet compatibility can be checked with a registry of component types that is consulted before connecting components. Run-time checking of messages[Scott and Finkel, 1988] is another possibility.

5.4 Modules and components compared

Modules are appropriate units for code that is reusable in building blocks for applications, but not in isolation. Examples are graphics devices such as buttons and scrollbars, which always form part of an application. These modules may be parameterized, e.g. for size and colour, using passed attributes or environment values. An analogy may be drawn with library routines; a sine routine is seldom useful without being part of a program.

The composition of modules is a larger module. A module must be combined with a parser to become a component. The PENGUIN method of composing components as a set of modules is superior to the traditional method of constructing a component with a big loop because it allows additional threads of interaction to to be added to a component without having to rewrite any code. Clashes and ambiguities in token values are detected by the compiler. The programmer doesn't even have to supply the parser because a standard parser works for any combination of modules.

Since there is one parser per component, a component has the property that it has exactly one point of computation at any instant. The pseudo-concurrency within a component is induced by the event-driven nature of PENGUIN programs. In PENGUIN programs, the times of arrival of tokens are the times at which a (possibly different) sub-parser can be activated. It is important that no semantic action be allowed to execute for an unbounded, or even large amount of time, otherwise the queues will fill up and the interactiveness of the component will suffer.

Activities that need to be run in parallel will require separate components. So long as the "real work" of a single component is of short duration, occasional pauses in the interactive response may be acceptable. For work that may take long, e.g. a lengthy computation, a friendlier interface would create a separate component to do the work. A protocol for sending information to, retrieving results from, and enquiring the status of this component would be needed. The user could proceed with other tasks while waiting for the component to finish the work.

The composition of components is a component. The internal "plumbing" between components is invisible from the outside of the "super-component", just as a user-defined program that invokes several other programs can be treated as a simple program in many operating systems. Components execute in parallel. Processes, as defined in many operating systems, are a natural implementation of components.

Another reason for designing a graphics application as a set of components is to limit the proliferation of design details and to allow parallel implementation of components. No matter how good the intentions of a designer may be, large systems cannot easily be constructed as monolithic systems. Once the boundaries between components have been drawn and the protocol agreed upon, the components may be implemented by separate teams. The gains in clarity must be balanced against the extra effort required to partition the system and define the protocols. PENGUIN helps this effort by encouraging the use of grammar notation in specifying the structure of communication between components.

5.5 The PENGUIN Model

The Seeheim model prescribes dialogue and application components in a program. These components can be realized as components as described in Section 5.2.

The Seeheim model can be generalized to the PENGUIN model, which allows an arbitrary number of components, connected via ports. If one recalls that the simplest case of a component is a single module and parser, then one sees that the presentation component also satisfies the criteria for components. The presentation uses a particular protocol, has action routines and manages resources, including, in this case, the physical devices such as the screen and keyboard.

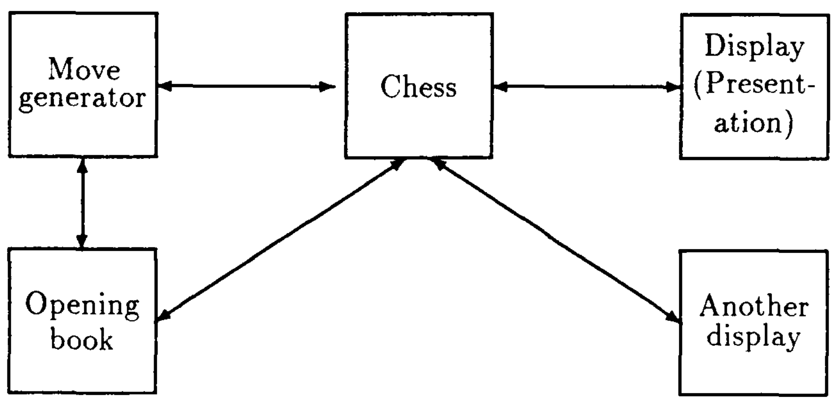

Figure 5.1: An example configuration in the PENGUIN component Model

Figure 5.1 is a hypothetical chess program, realized with communicating components. Note that non-PENGUIN connections to the outside world, e.g. to file systems, are not shown in the diagram.

The case for separate dialogue and application components in a program is not as strong as that for the existence of a presentation component. The original formulation of the Seeheim model envisaged that it would be possible to keep syntactic and semantic issues separate. "What class of feedback will be generated for input?" is a good question to ask when trying to determine the function of a component, i.e. lexical, syntactic or semantic. The chess program is an example that illustrates the different types of feedback. Typing an invalid key at the window is a lexical error; it can be detected by the presentation because the presentation knows the set of valid keys. Clicking on the right mouse button to confirm a quit command when the left button is the correct one to use is a syntactic error; the left button is valid at other times, but not in this particular command sequence. Trying to castle a king in check is a semantic error; it can only be discovered by the application component by checking the validity of the move, based on the locations of the pieces.

When a logical object is being manipulated, the syntax of the protocol is closely associated with the semantics of the object. It makes more sense to specify the syntax near the semantic specifications of a module rather than to gather all the syntactic and semantic portions of modules into their own components. Components that have been segregated by lexical, syntactic or semantic functions have *logical cohesion* or, at best, *communicational cohesion*[Yourdon and Constantine, 1979]. Both these types of cohesion are weaker than *functional cohesion*, that exhibited by PENGUIN components.

The presentation component, which normally is thought of as having syntactic functions, does in fact obey some protocol, albeit simple. The presentation is more properly considered as a manager of physical resources. That it normally performs lexical tasks is an artifact of its being first in the chain of communication from the user. Windowing systems, as presentation components, are constructed differently from clients applicat[illegible]s. We argue that this distinction is artificial. A windowing system should be constructed as a PENGUIN component and not as a completely different kind of program.

It is expected, therefore, that PENGUIN programs will not normally have separate dialogue and application components but may in fact have many components each of which combines dialogue and application functions (Figures 5.1 and 5.2). Our PENGUIN model allows syntactic and semantic portions within a component.

In this revised view, PENGUIN takes on not just the duties of dialogue control but also specifies to some extent the lexical and semantic tasks of the component. In the current implementation the terminal declarations are lexical specifications and the action routines are semantic actions. This dissertation began with a

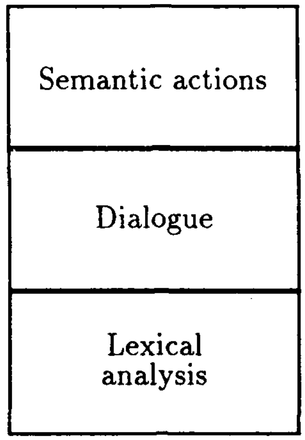

Figure 5.2: Substructure of a PENGUIN component

consideration of the syntactic, or dialogue portion of a graphics program, so the lexical and syntactic aspects were downplayed. However, in the light of the PENGUIN model, these must be considered as important a part of the interface as the dialogue is. In fact, the UIMS notion that the parts of the program that are for "interface" can be separated from those that are for "useful work" is untenable. In PENGUIN, decompositions are along the lines of responsibility for resources, not by application independent function.

An interesting consequence of the PENGUIN model is that the presentation component is interchangeable, provided that the presentation has sufficient capabilities, and the protocol between it and the other components is properly designed. This means that a graphics program can be made portable across different presentations at the expense of finding a common protocol for every presentation of interest.

6 Implementation

This chapter describes the implementation of the PENGUIN compiler, which reifies the design principles of Chapter 3.

6.1 PENGUIN Programs

The PENGUIN compiler is written in C++, with the aid of **bison** [Donnelly and Stallman, 1988], an LALR(1) compiler-compiler; **flex** [Paxson, 1989; Lesk and Schmidt, 1979], a lexical analyser generator and **g++** [Tiemann, 1989], the Free Software Foundation C++ compiler.

The unit of compilation is a PENGUIN module. A PENGUIN module comprises these sections: header, declarations, imports, private variables and the grammar. The header names the module. Declarations are typedefs, and symbol (terminal and non-terminal) declarations. Imports are declarations for linkage purposes and are prefixed with the name of an external module. Private variables are either module handle declarations or variable declarations. At the moment PENGUIN does not do full type checking so the types allowed are those in the target language, C++, with the restriction that the typename must be a single identifier. Typedefs soften this restriction by allowing constructed types to be given single identifier names.

The grammar section contains productions, which may have on the LHS non-terminals and on the RHS non-terminals, terminals or inline code. The production operator may be the sequential derivation operator, the parallel AND and parallel OR operators and the no-wait variants of the last two. Appendix A contains the PENGUIN grammar.

The compiler generates several types of output code. First come the preprocessor directives that include the appropriate external definitions (including those for imported symbols). Next come declarations of various types and structures used by symbols. Then follow data initializations for translation and parsing tables. Prediction of productions at run time is driven by the parsing tables. Finally, the

compiler generates a collection of recursive descent routines corresponding to the symbols and productions of the grammar. The last routine to be generated is the constructor for the class representing the module.

Each PENGUIN module results in an object file when compiled. This is linked with the run-time library, which provides the input routines, token dispatcher, coroutine scheduler and utility routines. A main program to start the collection of modules has to be provided by the programmer, but may eventually be provided in the run-time library.

The presentation component used for experimentation is the X11 Windowing System server and associated libraries. `Xlib`, a library of stub routines, comes with this server, and the input routines interface to this stub library. Tokens in PENGUIN terminology are called *events* in X11 terminology. PENGUIN takes advantage of the context that the X11 server attaches to each event. This context is the *window id* of the window from which the event originated. Other windowing systems could be interfaced to PENGUIN easily.

6.2 Generated code

This implementation of the PENGUIN compiler generates code in C++ [Stroustrup, 1986]. Two files are created when a PENGUIN module is compiled. The definition (`.h`) file contains declarations of the C++ class corresponding to the module and declarations of attribute structures for grammar symbols. The implementation (`.cc`) file contains the tables and code corresponding to the declarations and to the grammar.

6.2.1 Structure and class declarations

Structure declarations for symbol attributes are placed in the definition file for importation by other modules. The last item in the definition file is a declaration of the class implementing the module.

6.2.2 Tables

The prediction table is a matrix of lists indexed by non-terminal and terminal. Each list is a series of context offset and production number pairs.

A translation table is generated for converting from the terminal values that arrive from the outside world to the internal indices for the tables. This allows the parse table indices to be a dense set. For large, sparse, terminal sets, another translation mechanism, such as hash table lookup, could be substituted.

Bit sets are also generated to describe forks in which an alphabet is split. The parser checks the incoming terminal against these sets to decide which sub-parser gets the token.

The translation and prediction tables are not class members but are initialized data with file (static) scope and thus are visible only to routines in the accompanying module.

The `catchall` terminal is implemented by appending one entry to the terminal set. When the run time parser cannot find a match for a given terminal, it attempts a match using this pseudo-terminal.

6.2.3 Code

The first scheme for handling the augmented grammars envisaged having the compiler generate initialized tables which are used by an interpreter. The interpreter maintains a cactus stack corresponding to the parse history of the PENGUIN program. Forks create new branches in the cactus, and joins delete branches. Attribute passing is less straightforward. Either attribute types have to be restricted to a small set of types known to the PENGUIN compiler so that the offsets and sizes of members in attribute structures are known, or the compiler has to generate C++ routines for the copy actions to be called by the interpreter. Inline code also has to be turned into named routines. Considerable overhead is incurred if the ratio of inline code to grammar symbols is high.

For the reasons cited above, the coroutine method described in Chapter 4 is used. A PENGUIN module is turned into a set of coroutines. The cactus stack is implicit in the call history of the coroutines. Terminals in the grammar are turned into calls to a run time routine that expects input. Productions are turned into routines. Non-terminals are turned into calls to a run time routine that peeks at the next input token and uses that to decide between different productions. A coroutine package handles the thread switching.

PENGUIN makes the generated routines members of a derived class whose base class is `module`. Private variables are translated into class members. A constructor for the derived class is generated to initialize some pointers into the static tables.

Using the symbols **create** and **destroy** of an external module generate, in addition to the usual routine call, calls to a constructor or a destructor for that module, respectively.

Imported symbols cause the appropriate definition (`.h`) files to be included once only at the beginning of the implementation (`.cc`) file.

6.3 PENGUIN Run-time Library

We describe only the coroutine method run time algorithms here.

The heart of a PENGUIN component, the parser, is called from the main routine after initialization. There is one parser per component. This parser is passed the topmost module of a component. It is also passed a C++ class that provides input tokens.

Sub-parsers may be created in the course of parsing. Sub-parsers are created when a fork production is expanded, or an external symbol is expanded. The sub-parsers are implemented by coroutines.

When a token arrives, the sub-parser that might accept this token is ascertained through the sub-parser dictionary. Sub-parsers are entered into the dictionary when a fork production is expanded. When a parent waits for children productions to finish its entry is removed except in the case of the no-wait variants of fork productions, where the parent sub-parser remains in the dictionary.

A sub-parser must be entered into the dictionary under all the contexts that are carried by its start symbols. The compiler knows exactly how many contexts a symbol carries and passes a counted array of contexts to the run time fork routine so that the routine can put those contexts into the dictionary. The attribute structure is rearranged at compile time so that all the context attributes are contiguous in the attribute structure and can therefore be passed directly as an array.

Tokens that have no accepting sub-parsers are matched against the pseudo-terminal `catchall`. If this too fails to match then a run time error routine is called.

Sibling branches must be terminated when a fork is satisfied. To enable the termination code to gain control when a branch terminates, the branch is called indirectly through an enclosing run time routine. All child branches are passed a common data structure that keeps track of which branches are still active. The first OR branch or the last AND branch to terminate will clean up all its siblings. The parent routine may also need to be reactivated. A deficiency of the current implementation is that a fixed size stack is allocated when a branch is created. There is no bounds checking for the run time stack, nor can the stack grow on demand.

Branches that share a context are passed alphabet maps (bit sets) so that the dictionary lookup can correctly determine the sub-parser that will accept a token.

The coroutine dispatch code keeps a queue of sub-parsers that are ready to run. This queue is checked before reading input. One situation where there is more than one ready sub-parser is when a fork has just been executed and there are several sub-parsers that have not yet blocked on input. These sub-parsers must

all be executed as far as they can go, i.e. up to the appearance of a terminal. Only when this ready queue is exhausted does the parser read an input token.

The coroutine switch library is the only architecture specific part of the PENGUIN library. It contains one routine with a handful of assembly language statements and a couple of other routines that use architecture specific knowledge about the layout of the activation records on the stack. The only architecture supported at the moment is the Motorola 68020 [Motorola, Inc., 1985] series of CPUs but the code can be adapted for other architectures in a matter of hours or minutes. On the 68020, coroutine switching is done by saving the registers in a save area, then exchanging stack pointers. When the switch routine returns, execution resumes in a different sub-parser.

Context and other attribute values are passed as structures on the run time stack. Every terminal or non-terminal routine receives a structure whose members are the inherited and synthesized attributes of the item. The fields attached to items on productions are translated into the assignment statements involving attribute structure members.[1] Field names may also be used in inline code. These are also translated into names of attribute structure members.

Module handles are translated as class members or local variables of type *module pointer*. External references invoke the appropriate actions from the imported module. The create and destroy actions are special to the compiler and generate extra code to initialize the handle before calling the create action; and to free the handle after calling the destroy action.

Appendix B is a extended example showing input and generated code.

6.4 Input Management

The generated PENGUIN parser uses a simple interface to device specific input routines. These routines are provided by classes derived from the base input class. Since the PENGUIN runtime does not know in advance what device class will be linked with the parser, the base class contains, in addition to the terminal and context fields, pointers to device specific routines. These routine pointers are initialized by the constructor of the derived class.

```
/*
**      Definitions for base input class
*/

#ifndef         _baseinput_h
```

[1] We use the Eiffel [Meyer, 1988] convention that the declared members of a symbol are called attributes, and the identifiers used in productions are called fields.

```
#define        _baseinput_h    1

#include       <pg_types.h>

#define        ILL_TERM        (-1)
#define        ANY_TERM        (-1)

class baseinput;

typedef int    port;

typedef int (baseinput::*iproc)();

typedef int (baseinput::*iproc2)(port,...);

class baseinput
{
public:
        int             input_pending;
        int             val;
        context         ctx;
        iproc           get;
        iproc2          send;
        iproc           no_context;
};

#endif
```

The parser calls the device specific routines via the pointers. It expects the input pending flag, terminal value and terminal context fields to be filled in upon return from the input routine.

7 Experiences with PENGUIN

PENGUIN was used to program several pieces of local software with GUIs. Each of these will be discussed individually.

7.1 Case study: Pfig

`Xfig` is a line graphics editor. It presents the user with a canvas, upon which the user can draw geometric figures such as lines, squares and ellipses. The primary input device is the mouse. The keyboard is used for typing text objects on the canvas and entering file names in the data entry window. The mode of input is selected by a panel of simulated buttons. Some of the buttons are toggle switches, having two states. Some form groups in which only one button in a given group can be selected at any instant.

We have rewritten the user interface portions of `xfig` in PENGUIN, producing a functionally equivalent program called `pfig`. Figure 7.1 shows `pfig` in operation.

The hierarchy of the windows looks like this:

```
pfig
    canvas
    panel
        buttons
    message
    rulers
```

7.1.1 Observations

PENGUIN allows the programmer to modularize the `pfig` interface easily. `Pfig` also demonstrates the separation of the concepts of class and instance. Each

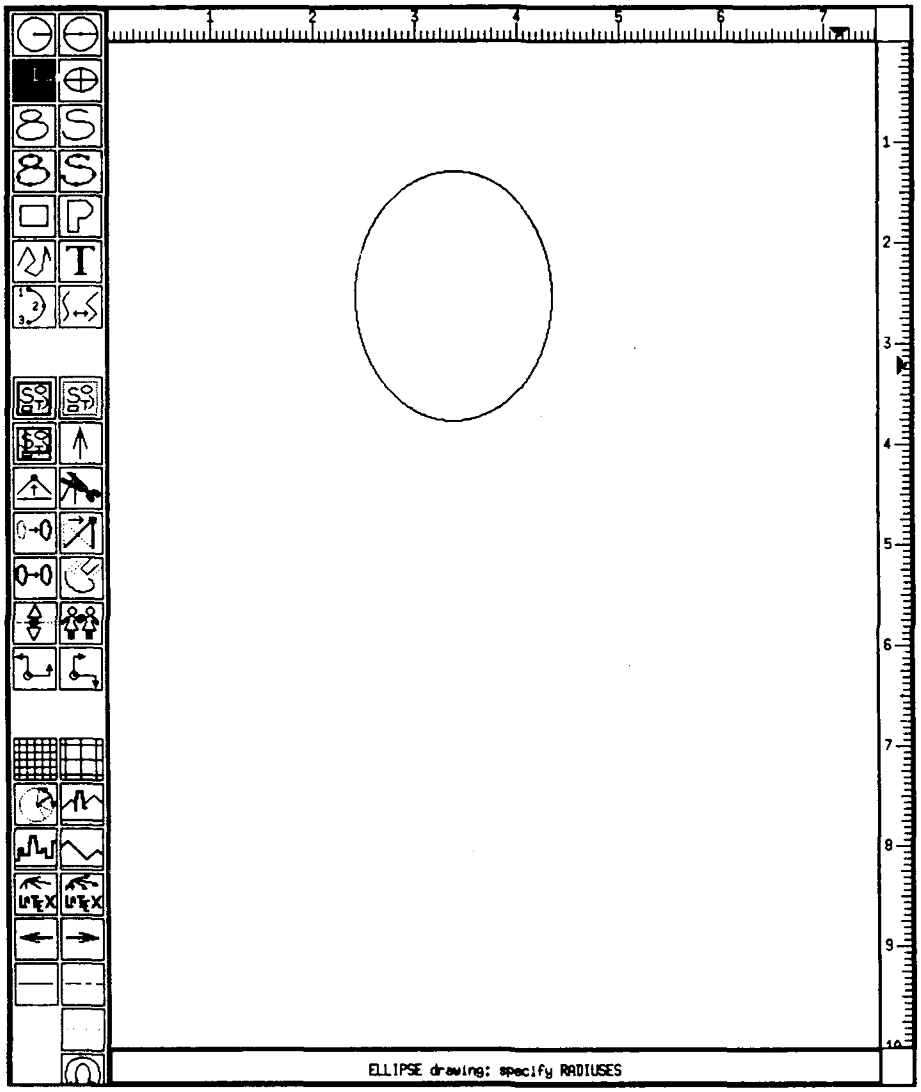

Figure 7.1: A `pfig` window

button shares the same code and parse tables but the legends on the buttons differ between buttons. More important, each button runs in a separate context. Events can be dispatched to the correct instance and cause the appropriate actions, in this case, sending a message to the panel informing it which button had been depressed. As each button controls its own private data, it is easy to allow each instance to track its own state, which is useful for making latching buttons.

It was easy to program each button to handle click tokens and exposure tokens. PENGUIN takes care of the alphabet split between different productions. Adding a production to handle focus tokens (cursor entering the button) to highlight the button was trivial, illustrating the ease of augmenting PENGUIN code that has been separated into modules.

PENGUIN makes possible a method for intermodule communication we call "token forwarding." Here is an example of how it it works. A panel comprises buttons. Each button may trigger a different action routine when clicked upon. Since the code template for the button is common to all buttons, it is not appropriate to put the call to the action routine in the button code. There are two traditional ways to deal with this situation. The first method is to give each button a unique index and have the code use the index to find the appropriate action. This requires a case statement or a global array somewhere. The second method is to pass a pointer to an action routine to each button (callbacks). PENGUIN make a third method possible: take the input token, change the context to one expected by the recipient and forward the token. In this example the buttons forward tokens to the parent panel. These tokens have the index of the button that was clicked. Each button instance saves this index from the attributes passed in at creation time. As far as the panel is concerned, a click event has happened at its window, even though the presentation does not deliver any. It is not necessary for modules to have a parent/child relationship for forwarding to work. Forwarding allows the pointers on the top and side rulers to track the movement of the cursor in the canvas window, and the canvas window is a sibling of the ruler windows.

7.1.2 Measurements

Measurements on `pfig` show that 687 lines of PENGUIN code expanded to 2100 lines of C++ code, a ratio of about 3:1. The `pfig` example replaced 2550 lines of C code with 2101 lines of PENGUIN, C++ and C code. This ratio becomes more favourable when one notes that some of the PENGUIN code for the abstract interactor modules (buttons, menus) provides functions that were previously provided by the X11 Toolkit. (`xfig` is 15000 lines large, but much of the code is windowing system independent.)

We estimate that the code generated by PENGUIN is about 50% more verbose

than what a programmer would write[1]. From this we conclude that the PENGUIN source is about a third to a half as verbose as the equivalent code a programmer would have to write in a conventional imperative language.

The X11 Toolkit is about 23000 lines of code while the PENGUIN runtime is about 1400 lines of code. However one must take into account the fact that the X11 Toolkit provides many more features than could be put into the PENGUIN run time library in the short time available. We estimate that about 5000 to 10000 lines of PENGUIN and C++ would suffice to recreate the X11 Toolkit. The X11 Toolkit suffers from having to implement object-oriented features such as instances and inheritance in plain C. It is also disadvantaged by having to use callbacks for semantic actions instead of being event-driven.

On a Sun-3, the text area of the **xfig** binary is 400 kbytes while the text area of `pfig` is only 300 kbytes. The difference is accounted for by the size of the X11 Toolkit. The initialized data and uninitialized data (bss) areas are similar in size.

7.2 Case study: Alarm clock

The second application is a simulated alarm clock with time and alarm displays, and buttons to set the display mode and the time of the alarm. This application demonstrates the handling of signals as events. The window hierarchy is simple:

```
wrapper
    clock
    alarm
    mode_button
    set_button
```

A run time library timer module sets up a handler for UNIX alarm signals. When the operating system delivers an alarm signal, the library translates it into a PENGUIN event. Since UNIX signals carry no information other than the fact that they occurred, an auxiliary queue stores a list of pending timer events. The library obtains the context and value of the token to create from the queue when a signal occurs.

The clock time and alarm time subwindows respond to exposure events, so they will redisplay the time when the window first appears or when it is unobscured. The clock subwindow schedules a timer event once a minute. When the minute event arrives, three things are done: the clock display is updated; current time is

[1] It is little unrealistic to assume that a programmer could generate the prediction tables by hand though.

compared with the alarm time and if they match, the clock beeps; and another timer event is scheduled for a minute in the future.

Because the alarm clock was coded from scratch, there is no old implementation to compare against. Some indication of the degree of programming help provided by the compiler can be seen from these statistics: 312 lines of PENGUIN code generated 1129 lines of C++ code. Another 232 lines of auxiliary routines in C++ were needed. The run time library is identical to that used by `pfig`, except for the addition of the timer module. There are 29 productions, of which 5 are forks.

Our subjective impression is that the clock would have been noticeably harder to write without PENGUIN. The clock code would also have been less portable. A clock written with traditional GUI libraries would have contained calls to environment dependent services to obtain the alarm signals. In the PENGUIN world all signals from the outside world are dealt with as events; the run time library provides the translation. With the timer code in the run time library, future programs requiring timer events would be easier to write. When porting programs requiring timing services to a non-UNIX environment only the run time library timer routine would need to be rewritten.

7.3 Case study: Graphical printer interface

`Print` is a program to submit jobs to the printer queue and notify the user when the job is complete. It demonstrates the use of components to divide up work. There are two components: one deals with the user interface and job submission; the second monitors the printer queue, notifying the first component when a job completes. This design allows the user to continue to submit iobs without waiting for the first to finish.

The module hierarchy of the print program is even simpler than that of the alarm clock:

```
wrapper
    slot
```

The screen area of the print tool is divided into a number of slots, each of which tracks the status of one file. The number of slots is fixed at compile time for simplicity. Initially all slots are empty. When the user types in the name of a file and confirms the request with a final carriage return, the job is submitted to the printer queue. The monitor component is notified. After some time, when the file has been printed, the monitor component sends an event to the print tool.

which then highlights the file name to draw the user's attention. The slot is then free for another print request. If the request fails because the file name is invalid or the file does not exist, the name disappears from the slot shortly after submission.

The protocol for intercomponent communication is simple. A request packet is sent to the monitor component for every file to be printed. Eventually a reply packet is returned, indicating either that the file did not appear on the queue (which is an error condition), or that the file has completed printing. It is neither necessary nor desirable to make the monitor component respond to status queries; the print tool already knows the status of the each request and can answer queries directly.

The print tool continues to maintain the appearance of the window while printing occurs in parallel. The print tool demonstrates how operations that require an indefinite amount of time to complete may be delegated to an auxiliary component, allowing the user interface to remain reactive. The same design principle can be applied to programs with a heavy computational component to keep the GUI portion interactive.

Measurements show that the print tool required 159 lines of PENGUIN code to produce 484 lines of C++ code. An additional 241 lines of auxiliary routines in C++ were required. There are 12 productions, of which 2 are forks. The run time library is the same as that used by `pfig` except for the addition of a module to handle intercomponent communication.

7.4 Observations

Multiple threads and context based token dispatch work as expected. The PENGUIN run-time library takes care of the ugly details of thread creation and cleanup, leaving the grammar and code uncluttered.

Windowing specific code is concentrated in the run-time library. It would not be difficult to rewrite the library to use a different windowing system. Portability requires foresight when designing the application and libraries; windowing system dependencies must be kept out of application code and dealt with in the run time library interface.

Some difficulty was encountered with lexical classification. PENGUIN expects a flat token alphabet. Real world token sources may have hierarchical alphabets where one field of the token selects variants for the other fields, e.g. button: left, middle, right; key: any key. For the moment, this is swept under the rug by requiring each application to provide a lexical translation routine.

Attribute handling in PENGUIN is clumsy. This is partly the fault of the implementation, which uses structures to pass all the attributes of a symbol. This means that assignments have to be generated prior to and after the routine

call corresponding to an item. In retrospect it might have been better to use a conventional argument list instead of passing attribute structures. However this method has its own drawbacks: unassigned attributes and synthesized attributes need special handling.

This particular implementation of attribute handling also caused problems when coding the part of the main program that sets up the initial set of modules. The code for initializing all the attribute structures was clumsy.

Converting an existing application to use PENGUIN was more time consuming than expected. The main hurdle is to retain the original semantics while changing the dialogue structure. `Xfig` was originally written in C and interfacing to C++ presented some difficulties. The final binary combines object code generated by the C and the C++ compilers. The original assumption that C and C++ object modules could be easily mixed proved not to be true. C++ is a more strongly type-checked language than C and it was necessary to specify all the entry points in the C object modules called from C++. In retrospect, it would have been cleaner to rewrite the original C code in C++. However, the original C code contains dubious coding practices so this is not an easy solution either.

7.5 Evaluation of PENGUIN against goals

7.5.1 Specification

PENGUIN's event-driven model and modules force the programmer to specify *all* the possible tokens that could arrive at a module. It is therefore easy to see all the possible sequences triggered by input. In contrast, the polling method requires the reader to untangle the execution paths taken by tokens belonging to various contexts. The other traditional method, using callbacks, requires elaborate initialization of data structures containing pointers to callback action routines. The body of the action routines may be quite far from where the module is created. The PENGUIN method keeps action routines close to the grammar specification.

In addition, the compiler generates, upon request, a prettyprinted list of productions in the grammar which allows the programmer to see at a glance the possible sequences of events. Here is the listing for the button module. It is the skeleton of the `pfig` grammar.

```
Start productions
0. create => _INLINE_
1. destroy => _INLINE_
2. S &> work repaint
3. work => click*
4. repaint => events*
Productions:
0. click => left _INLINE_
1. events => expose _INLINE_
2. events => pointerin _INLINE_
3. events => pointerout _INLINE_
```

7.5.2 Translation

PENGUIN takes a module specification and generates code that the programmer would have had to write. Prediction tables, which are normally tedious to compute by hand, are computed by PENGUIN. Appendix B exhibits the PENGUIN code for the canvas module in `xfig` as well the generated code, for comparison.

PENGUIN's input language allowed us to structure an application as a set of reactive modules. The conceptual simplification introduced is significant. Firstly, no time precedence relationships can be assumed across productions except when one production predicts another; precedences only apply within productions. Secondly, the private variables within each instance of a module are local resources; access to global variables requires explicit importation or the use of external routines. These two conditions enforce a programming discipline that requires the programmer to ensure that each production leaves resources in a consistent state when a production finishes. Effects of semantic actions are as localized as possible. Reasoning about the module as a whole can be done by examining one production at a time, in particular by noting which sequence of input events will activate each production. Reasoning about the component as a whole can be done by examining the actions contained in and resources managed by each module.

PENGUIN's run time library manages the creation and destruction of threads. It is very pleasant not have to manage threads oneself and to be able to enhance the behaviour of existing code by adding more productions. In all the examples above, fewer lines were needed in PENGUIN than if the standard Toolkit library had been used.

7.5.3 Portability

Modules can be written to deal with abstract events, and the binding of the events to the presentation (in the case of **pfig**, the windowing system) can be pushed into the run time library. The **pfig** example in Appendix B shows the PENGUIN code for a button module containing calls to run time routines to handle events. It would be simple to retarget run time library code for say the SunView environment.

Note that callbacks do not work in languages where procedure variables are not supported. PENGUIN's event driven model is more portable in this respect.

Only about 20 lines of C and assembler in the run time library for coroutine support are machine specific. Only Sun-3s are supported at the moment, but other machines would be trivial to add.

It is instructive to compare the construction of pfig with an earlier software effort in which the original version of the graphical editor, fig, was ported from the SunView windowing system to the X windowing system over a period of two months in the author's spare time, resulting in **xfig**. The work was mechanical, boring and tedious. The same windowing services were needed by the editor in both environments: the major differences were in the syntax of the routine calls and the data structures involved. Fortunately, by judicious use of C preprocessor and compiler features, a decent translation resulted. This experience lends support to the assertion that GUI programs written in a higher level notation such as PENGUIN can be ported with less effort to a variety of GUI platforms than would be needed for translation between windowing systems. Translation efforts such as that from fig to xfig can be avoided if the software is written in PENGUIN in the first place. Such GUI programs would be written to handle abstract events and use generic data structures. The PENGUIN run time library would handle most of the graphical environment dependencies.

7.5.4 Reusability

Two ways of reusing a PENGUIN module are (1) to create a module that can be parameterized for various purposes and (2) to change the source of an existing module for a slightly different purpose.

A good example of a reusable module of the first type is the button module in **pfig**. Not only does this module serve as the basic unit in the control panel but it also serves as the basic item in a popup menu, even though panel buttons are square with pictures on them and the menu items are rectangular with text legends. The panel buttons do not highlight when the cursor enters the button while the menu items highlight. Each button is customized by passing a data structure that describes the characteristics of the button required. This module

can easily be reused in other windowing applications. Many of the other modules created for `pfig` such as menus and panels are reusable in this way too.

Conventional windowing systems do not cater to this type of reusability as well as PENGUIN does because support for attribute passing through a hierarchy of modules is poor. This is a consequence of grafting a UI notation on top of a conventional language instead of providing compiler support for attribute flow.

An example of a reusable module of the second type is the wrapper module from the tools in the case studies. There are only minor differences between the different versions, inspite of the different applications, because the purpose is similar in each version: create all the descendant windows in the application and manage the contexts created. It was only a matter of a few minutes of editing to transform a wrapper module written for `pfig` to one suitable for `alarm`.

While in principle modules written for conventional windowing systems can be reused in the same way, in practice the low level of programming detail that gets introduced into such modules make it hard to distangle the windowing system independent tasks of a module from the incidental minutiae required by the particular environment. PENGUIN encourages the programmer to push such concerns into the run time library or at least into subroutines called from PENGUIN modules.

7.6 Problems

Although grammars made the control structure of the program clear, the number of levels of productions used were not as deep as we had expected. Part of the reason may be because programmers hesitate to make deeply nested structures with the traditional methodology. Only more experience with using PENGUIN to write applications from the ground up can tell.

Grammars replaced less of the traditional program than we had hoped. A lot of the work of a graphics program is concerned with error checking and other semantic work which has to be done whatever control structure is imposed.

PENGUIN's context features were not used as much as we had expected. Most of the time only a single context was used in a module. We attribute this partly to lack of experimentation with these new features and how they might be used in new programs.

PENGUIN should not have bothered with controlling the flow of non-context attributes. It was a waste of time to separate inherited and synthesized attributes. One small argument in favour of retaining the separation is that it keeps the interpretive approach to run-time parsing (Chapter 6) feasible; an interpretive parser needs to know the direction of attribute flow to generate the appropriate calls to copy attributes between stack frames; in a native parser the programmer is

responsible for managing the flow of non-context attributes. However the superiority of the native code method is too strong. PENGUIN should have just allowed the programmer to use whatever host language parameter passing capability was available. Knowledge about the context attributes is still required to enforce context uniqueness.

There were some features that could not be programmed in PENGUIN such as starting up a large but finite number of threads. For this one needs a counter, which can only be done by an escape to the host language. This situation occurs often, so either PENGUIN should provide some iterative construct in the future, or some more structured way of using host language features should be allowed.

Initially PENGUIN programming seemed strange because of the declarative nature of the grammar. It became more natural with use. To win acceptance with more users, more experience is needed to see which parts of the language could be made more natural.

8 Conclusion

In this chapter we summarize the contributions of PENGUIN, suggest directions for future work on PENGUIN, and prognosticate a little on the future of GUIs.

8.1 Contributions

We have described the design philosophy and underlying theory of PENGUIN. Our experiences with PENGUIN confirm that compiler technology can ease the task of programming GUIs. This is an important result because in the future more and more computers will be used by non-specialists. Future GUIs will stress ease of use with more effective human-computer interfaces. Effective interfaces require more programming effort. If software productivity is not to suffer, then programmers must have more sophisticated tools.

PENGUIN is an important advance in GUI technology for the following reasons:

Expressive notation Among the various candidate notations for GUI dialogues, grammars are the most expressive and concise. Our experience with dialogues written in PENGUIN bears this out. Grammars are generally a third to a half of the size of the code they replace and provide a clear skeleton structure for the code. PENGUIN's fork operators have proved to be adequate to deal with practical dialogues.

Code generation PENGUIN is a tool for automating much of GUI dialogue construction. Fork productions provide a high-level abstraction for multiple threads; the compiler maps the grammar productions into multi-threaded code. The compiler and run time system ensure the correct dispatch of tokens.

Improved portability The PENGUIN run time library isolates the programmer from environmental dependencies. Routines for event input have only to be

written once for each environment and thereafter are available to all PENGUIN programs, amortizing the programming effort over many applications. The uniform event model of PENGUIN treats all input data, signals and exceptions as events. Since event handling is one of the more environment dependent parts of a GUI program, pushing these functions into the run time library makes PENGUIN programs easy to port.

Thread management GUIs require support for multi-threaded behaviour. PENGUIN separates thread management concerns from sequential programming. The runtime library manages threads, thereby simplifying the programmer's job. PENGUIN's grammar notation and context-based dispatch are essential to a practical and efficient implementation of multiple threads.

Reactive execution PENGUIN's reactive execution model is a better fit to GUI dialogues than the approaches taken with imperative languages: polling and callback routines. In PENGUIN notation, execution is driven by input rather than by control flow structures. This allows the programmer to write a module as a set of input acceptors. The PENGUIN event notation frees the programmer from spurious concerns that have been introduced by forcing event-driven programming into the sequential programming model. Sequentiality needs only be considered within a sub-parser.

Delegation by redirection The quantization of input into events and the use of sub-parsers to deal with separate threads allows a new way of delegating responsibility to a different sub-parser: redirection of events. This mechanism is general and does not depend on any hierarchy of PENGUIN modules in a program.

Component model The PENGUIN intercomponent connection model provides a framework for building complex applications as a network of independently executing components. The grammars of the components formally specify the protocols that the components will use to communicate with each other. The traditional Seeheim model of partitioning an application by lexical, syntactic and semantic responsibilities gives way to a model in which components manage resources and communicate across logical application boundaries.

In summary, the use of traditional programming methodologies results in GUI programs that have unnatural structure. Although the dialogue is but a part of graphical applications, it is the critical heart of such software.

8.2 Desirable PENGUIN enhancements

PENGUIN could easily be improved to make usage more convenient.

More automation The compiler could do more work for the programmer. If a multi-way fork is needed, the programmer currently has to provide auxiliary symbols because only non-terminals are allowed on the right hand side of a fork production.

```
A &> B C
B => t*
C => X Y Z
```

The compiler should allow the programmer to write

```
A &> t* { X Y Z }
```

by creating auxiliary symbols as needed in the generated code.

Lexical issues The terminal classification scheme is too rigid. It is assumed that each terminal in the grammar represents exactly one element of the token alphabet. An enhancement to PENGUIN would allow the programmer to classify tokens using more than one attribute in the terminal and would let the compiler take over more of the work of the programmer. For example, the programmer might specify that a range of alphabet elements map to a terminal in the grammar, e.g. all keyboard tokens are collectively called a key terminal and the key value is passed to semantic routines in a separate attribute.

At this moment this limitation is bypassed by performing translation in the input routine. This augmentation would not alter the context-free property of the grammar.

8.3 Future research issues

In the long term, improvements to PENGUIN will require investigation into several open research issues. PENGUIN can be improved by providing general and powerful mechanisms that will increase the expressivity of the notation and improve the reusability of code. PENGUIN should also be made to coexist better with other programming tools.

Inheritance Inheritance is a way of allowing common features to be factored out in a common ancestor. For example, a button module may need to be specialized to make a locking button. Inheritance would make PENGUIN grammars more reusable. An interesting research issue would be to extend inheritance to productions. Current object-oriented languages define inheritance only for data and procedures.

Grammars as parameters Grammars are static global objects. Grammars might be more useful if they could be passed as parameters, giving modules whose behaviour could be altered dynamically.

Exceptions Sub-parser resources that are managed by the PENGUIN run time are automatically recovered at the end of a fork. Resources that are explicitly allocated by programmers within sub-parsers will be lost. There should be a way to execute programmer specified code that would free these resources prior to coalescing the branches of a fork.

Language environment PENGUIN is not well integrated with existing language environments. Some things cannot be done easily in PENGUIN but the mechanism for escaping to the host language is awkward. For example, the syntax for declaring private variables is restrictive and makes it difficult to declare the full range of data types allowed in the host language. Another example is the need for an unnatural notation to signal to the compiler references to attributes in inline code. Ideally, PENGUIN should handle token dispatch, flow of control, and thread management, but allow the programmer to use the host language to achieve other goals.

Components Intercomponent compatibility has been formally defined in Chapter 5, together with possible implementation techniques but experience is needed to determine which techniques are useful.

Other problem domains PENGUIN is a language for describing reactive execution in general. Many other problem domains require programs that accept input from multiple sources, using context to dispatch to threads. Although context is normally associated with windows in a graphical user interface environment, it could equally well be used to identify one of several sensors in a real-time embedded program or one of several communicants in a network protocol program. PENGUIN may prove to be a useful tool for constructing such software.

8.4 Conclusion

Although we have shown that PENGUIN is a superior tool for the construction of GUI dialogues, there is little impetus within the GUI community to abandon current dialogue construction practices. There are several reasons for this:

- Current GUI programming praxis is conservative, being concerned with control over appearance and with the standardization of interfaces, as shown by the emergence of "look and feel" toolkits for windowing systems. Current

GUIs do not exhibit a wide variety of interaction styles. A small selection of buttons, menus, scrollbars and viewports suffices for most current applications. This reflects the conservative nature of the GUI market. Users who have grown accustomed to certain styles of interaction are reluctant to switch to new interaction techniques without a guarantee of adequate return on learning effort.

- Current practice also reflects insufficient experience with novel I/O devices. The standard configuration of screen, keyboard and mouse encompasses the overwhelming majority of graphical interfaces. The current GUI practice is to concentrate all device specific code for standard configurations in a single, monolithic presentation component. This methodology is adequate for standard configurations but does not permit additional interaction channels such as sound to be added easily. With current practice, the presentation component cannot be easily modified or replaced by the programmer. PENGUIN's model allows as many presentation components as needed to communicate with devices to be created.

- Traditional programming languages have a great deal of inertia. For example, in the X11 windowing system, toolkits would clearly have benefited from a modern language such as C++, instead of C, but C continued to be used because of concerns over the availability of C++ compilers for a wide range of platforms.

In the long run, appropriate methodologies such as PENGUIN will prevail because advances in GUI hardware and software will necessitate the adoption of better tools. Appearance and standardization issues do need to be resolved but in the longer term, advanced interfaces for use with sophisticated hardware will emerge. Devices such as wired gloves and head-up displays will come down in cost with development. These devices will require new interaction techniques, different from those developed for windowing displays. Even with windowing displays, increases in hardware power will permit more natural interaction techniques such as animated interfaces, which cannot be easily catered for by current GUI toolkit technology, but require special programming effort. A structural engineer may experiment with a framework by manipulating objects on a screen and then requesting an almost instant analysis of the structure. When these advanced interaction techniques come into wider use, current interface techniques and software tools will prove inadequate.

The history of progress in Human-Computer Interfaces can be read as a continual search for improvement in the bandwidth of communication channels between the machine and user. A person directly manipulating a physical object gets all the information the senses can absorb. A person sitting at a computer

terminal must maintain a mental model of the objects inside the computer system. Textual user interfaces require the user to retain more context than GUIs do. Textual interfaces were necessitated by the narrow bandwidth communication channels we once had. GUIs provide wider bandwidth for interaction. Techniques such direct manipulation [Shneiderman, 1982] can take advantage of this increase in bandwidth but the programming tools we have are wanting. In the GUI world, information entering a program does not come from a single source only, nor will users be willing to react to a program only when bidden to do so by the program. Our tools must support these freedoms permitted by GUIs.

A PENGUIN Language Manual

A.1 Introduction

This manual describes the PENGUIN language. The manual is organized as follows:

1. Introduction
2. Lexical Conventions
3. Module sections
4. Declarations
5. Storage
6. Grammar

A.1.1 Syntax notation

In the syntax notation used in this manual, syntactic categories are indicated by an *italic* face, and literal words and characters by a `typewriter` face. Alternatives are listed on separate lines. Zero or more instances of an item are indicated by a superscripted asterisk:

$item^*$

An optional terminal or non-terminal symbol is indicated by the subscript "*opt*," so that

{ $expression_{opt}$ }

indicates an optional expression enclosed in braces.

A.2 Lexical Conventions

The unit of translation is a PENGUIN *module*. Currently this is synonymous with *file*.

A.2.1 Tokens

There are six types of tokens:

- Identifiers
- Keywords
- Constants
- Strings
- Operators
- Other separators.

Blanks, horizontal tabs, new-lines, formfeeds, and comments (collectively, "white space"), as described below, are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to be the longest string of characters that could possibly constitute a token.

A.2.2 Comments

The characters `/*` start a comment, which terminates with the characters `*/`. These comments do not nest. The characters `//` start a comment, which terminates at the end of the line on which they occur. The comment characters `//`, `/*`, and `*/` have no special meaning within a `//` comment and are treated just like other characters. Similarly, the comment characters `//` and `/*` have no special meaning within a `/*` comment.

A.2.3 Identifiers

An identifier is an arbitrarily long sequence of letters, underscores and digits. The first character must be a letter. Upper- and lower-case letters are different. All characters are significant.

A.2.4 Keywords and Operators

The following identifiers are reserved for use as keywords, and may not be used otherwise:

```
const external grammar include module nonterm private returns
        terminal typedef
```

PENGUIN programs use the following characters as operators or for punctuation:

```
( ) * + - : , . $ @
```

and the following character combinations are used as operators:

```
=> &> |> &: |:
```

Each is a single token.

A.2.5 Literals

There are several kinds of literal constants:

literal:

integer-constant
character-constant
string

Integer constants

An integer constant consists of a sequence of digits and is taken to be decimal. The range of representable integers is machine dependent but may be assumed to be signed 32 bits in current implementations.

Character constants

A character constant is one or more characters enclosed in single quotes, as in `'x'`. Single character constants have type **char**.

Certain non-graphic characters, the single quote, the double quote, and the backslash, may be represented according to the following table of escape sequences:

new-line	NL (LF)	`\n`
horizontal tab	HT	`\t`
vertical tab	VT	`\v`
backspace	BS	`\b`
carriage return	CR	`\r`
form feed	FF	`\f`
alert	BEL	`\a`
backslash	\	`\\`
single quote	'	`\'`
double quote	"	`\"`
octal number	*ooo*	`\ooo`
hex number	*hhh*	`\xhhh`

If the character following the backslash is not one of those specified, the behaviour is undefined. An escape sequence specifies a single character.

The escape *ooo* consists of the backslash followed by one, two, or three octal digits that are taken to specify the value of the desired character. The escape \x*hhh* consists of the backslash followed by x followed by a sequence of hexadecimal digits that are taken to specify the value of the desired character. A sequence of octal or hexadecimal digits is terminated by the first character that is not an octal digit or a hexadecimal digit, respectively.

Strings

A string literal is a sequence of characters (as defined in § A.2.5) surrounded by double quotes, as in "...". Within a string, the double quote character must be preceded by a \.

A.3 Module sections

A module comprises the following sections, some of which are optional:

module:

  header $declarations_{opt}$ $externals_{opt}$ $private_{opt}$ *grammar*

A.3.1 Header

The header provides a name for the module. The name should be the same as the filename body in the current implementation.

header:

  `module` *identifier*

A.3.2 Declarations

A declaration introduces one or more names into a module and defines the type of a name.

declaration:

typedef
terminal declaration
nonterminal declaration
insert
include

Terminals and non-terminals are collectively called *items*.

Typedef

typedef:

`typedef` *identifier* = *definition* ;

A typedef introduces an abbreviation for a type definition. The definition is any type declaration that is valid in the target language, currently C++.

Terminal declaration

terminal declaration:

`terminal` *terminal-list* ;

terminal-list:

terminal
terminal , *terminal-list*

terminal:

identifier $attributes_{opt}$ $value_{opt}$

value:

= *integer-constant*
= *character-constant*

A terminal declaration names tokens accepted by the parser. A value can be associated with a terminal with the optional value clause. In the absence of a value clause, a terminal is assigned a value one greater than the last terminal value. If the first terminal does not have a value clause, the value defaults to 0.

Attributes may be associated with a terminal. See § A.3.3.

Nonterminal declaration

nonterminal declaration:

    nonterminal *nonterminal-list* ;

nonterminal-list:

    nonterminal
    nonterminal , *nonterminal-list*

nonterminal:

    identifier $attributes_{opt}$

A nonterminal declaration names nonterminals and associates attributes with those names. See § A.3.3.

Inserts

Inserts are used to pass lines uninterpreted to the target language processor.

insert:

    $(*arbitrary text* **$)**

Includes

An include directive causes the compiler to interpolate the contents of the named file after the include directive.

include:

    include *string* ;

A.3.3 Externals

externals:

    external *modulename declarations*

An external section allows declarations to name non-terminals that are imported from other modules and associates attributes with those names. The first identifier must be the name of a module and the second identifier a non-terminal from that module.

Attributes

attributes:

($inherited_{opt}$ $synthesized_{opt}$)

inherited:

attribute-list

synthesized:

| *attribute-list*

attribute-list:

attribute

attribute , *attribute-list*

attribute:

typename attributename

typename:

identifier

attributename:

identifier

Attributes name regions of storage that are used to pass information between symbols. The scope of an attribute name is local to the associated item, all references must be qualified.

A.4 Private data

private:

`private` *private-declaration**

private-declaration:

typename variablename ;

`module` *modulename variablename* ;

typename:

identifier

modulename:

identifier

variablename:

identifier

A private declaration names regions of storage that are private to a module. Each instance of a module is assigned a separate region of storage.

A.5 Grammar

grammar-section:

grammar *production**

production:

lhs local-list operator rhs synthesized$_{opt}$;

lhs:

identifier

local-list:

*storage-declaration**

operator:

=>
&>
|>
&:
|:

rhs:

*item**

synthesized:

returns *fields*

item:

identifier fields$_{opt}$
inline-code

inline-code:

$(*arbitrary-statements* **$)**

fields:

(*field-list*)

field-list:

field
field , *field-list*

field:

variablename
@ *itemname* . *fieldname*
@ . *fieldname*
@ *integer-constant* . *fieldname*
expression
$ *envname*

variablename:

identifer

itemname:

identifier

fieldname:

identifer

envname:

identifier

The grammar section lists the productions contained in the module. A production specifies the sets of sequences of terminals, non-terminals and inline code that are accepted as the expansion of the LHS non-terminal.

Field lists specify inherited attributes or expressions to be copied into attributes of non-terminals. Attribute flow between items in a production must be from left to right. Inline code may name attributes of items to the left in the production. The optional attribute return list specifies synthesized attributes that are returned by the LHS non-terminal to the superior production.

The local list comprises declarations of variables that are private to the production.

A.6 Execution model

A PENGUIN component must contain at least one module. When execution begins in a component, at least one instance of a module must be created. A instance of a module is created by predicting the special non-terminal `create` of the module. This operation has the side effect of initializing a handle to the instance of the module created. The special non-terminal `destroy` recycles the handle for use by other created instances.

A production becomes active when the LHS non-terminal is predicted by some other production (in the same module or in an external module), or when the initial start-up routine predicts the LHS non-terminal. A production terminates when a valid sequence has been accepted. Productions may predict terminals, which must be matched by input tokens. Productions may predict inline code. The specified code is executed when that position in the production is reached.

When a normal production becomes active, it predicts the sequence that is accepted by the RHS symbols. In the case of fork productions, indicated by the use of operators other than `=>`, partial sequences for some of the RHS symbols may be accepted. The `&>` operator accepts a sequence that is an interleaving of all sequences accepted by the RHS symbols. The `|>` operator accepts a sequence that is an interleaving of a sequence accepted by one right hand side symbol and sub-sequences accepted by all of the other RHS symbols. For both operators, the

production predicting the LHS symbol is suspended until the LHS symbol is accepted. The **&**: and **|**: operators are defined similarly, except that the production predicting the LHS symbol is not suspended, but continues to accept sequences to the right of that symbol.

Input tokens arrive from other components and are read from the urgent queue, or the normal queue, if the urgent queue is empty. Tokens are tuples containing at least two fields, value and context. Both queues are ordered by time of arrival. Each token is dispatched to the production awaiting a terminal of that value and context. Tokens that cannot be dispatched are sent to a run time error handler. Normal and urgent tokens are sent to other components by calling the appropriate run time library routine.

No time precedences are implied by the order in which the productions are listed. The only time precedences are those implied by the order of items within a production and the prediction of non-terminals from a production.

B Extended Example

This appendix contains the complete source code for the user interface portion of the pfig line graphics editor. The hierarchy of the modules is shown in Chapter 7. This appendix is divided into three sections: (1) the PENGUIN modules comprising the GUI, (2) the main program, which also contains the glue routines to the X windowing system, and (3) an example of code generated from a PENGUIN module.

B.1 PENGUIN modules

Wrapper module

The first module we exhibit is the wrapper module. The wrapper window is not sensitive to any events but encloses all its descendant windows.

```
//
//      wrapper window for pfig
//
module pwrapper

$(
#include        <std.h>
#include        <stream.h>
#include        <Xinput.h>
#include        "pfig.h"
#include        "resources.h"
$)
        include "pwrapper.pi";

external ppanel
        include "ppanel.pi";
external pcanvas
        include "pcanvas.pi";
external pruler
        include "pruler.pi";
```

```
external pmessage
        include "pmessage.pi";

private
        module ppanel P;
        module pmessage M;
        module pcanvas C;
        module pruler HR;
        module pruler VR;
$(
        extern "C" void setup_rulers();
$)
```

These are preliminary setup declarations. The includes bracketed by `$(` and `$)` are passed through to the C++ compiler. The PENGUIN includes contain the attribute declarations of external non-terminals.

```
grammar

create
        Window          w;
        =>
        $(
                w = xi->create_enable_context(@.wrapper_str);
                cerr << "Wrapper window " << hex(w) << "\n";
                @.message_str->parent = @.hruler_str->parent =
                        @.vruler_str->parent = @.canvas_str->parent =
                        @.panel_str->parent = w;
        $)
        P:create(@.panel_str,@.button_str,@.nbuttons)
        HR:create(@.hruler_str)
        VR:create(@.vruler_str)
        C:create(@.canvas_str,@.menu_str,@.item_str,@.nitems,
                @HR:create.ctx1,@VR:create.ctx1)
        $(
                topruler_pixwin = trswfd = @HR:create.ctx1;
                sideruler_pixwin = srswfd = @VR:create.ctx1;
                setup_rulers(@.hruler_str->bgc,@.hruler_str->fgc);
        $)
        M:create(@.message_str)
        returns (w, @P:create.ctx1, @C:create.ctx1,
                @HR:create.ctx1, @VR:create.ctx1, @M:create.ctx1);

destroy =>
        $(
                cerr << "Wrapper destroy\n";
        $);
```

The create non-terminal calls a run time library routine to obtain a context for the wrapper window and saves this context in the window structures of its descendant windows: a panel, a message area, a canvas (drawing) area, and two rulers. Next it predicts the create non-terminals of each of its children modules, passing along the window structures inherited from the main routine. For compatibility reasons, some global variables from the C routines need to be initialized. Finally, all the new contexts are returned.

```
S =>
        $(
                cerr << "Wrapper S ctx " << hex(@.ctx) << "\n";
        $)
        start(@.ctx1, @.nbuttons, @.ctx2, @.ctx3, @.ctx4, @.ctx5, @.dpy);

start   &>
        P:S(@.nbuttons,@.ctx1)
        C:S(@.ctx2)
        HR:S(@.ctx3)
        VR:S(@.ctx4)
        M:S(@.ctx5);
```

The S non-terminal is called from the main routine after the creation phase to start the program running. All this non-terminal has to do is predict all the S non-terminals of its children.

Canvas module

The canvas window provides the drawing surface on which the user creates line graphics.

```
// canvas window for pfig
module pcanvas

$(
#include        <std.h>
#include        <stream.h>
#include        <Xinput.h>
#include        "pfig.h"
#include        "resources.h"
$)

        include "pcanvas.pi";
        terminal expose = 'e', key = 'k',
                left = 'l', middle = 'm', right = 'r', pointer = 'p';
        nonterm repaint_loop, work_loop;
```

```
        nonterm repaint, work;

external pmenu
        include "pmenu.pi";

private
        module pmenu M;
        int menu_count;
        Wstruct *m_info;
        context mctx;           // of menu
        context hctx;           // of hruler
        context vctx;           // of vruler
$(
        extern "C" void redisplay_canvas();
        extern "C" void pointer_move();
        extern "C" void key_press();
        extern "C" void null_proc();
        extern void (*canvas_leftbut_proc)(int, int);
        extern void (*canvas_middlebut_proc)(int, int);
        extern void (*canvas_locmove_proc)(int, int);
        extern void (*canvas_kbd_proc)(int, int);
$)
```

These preliminaries differ from those of the wrapper window in that terminals are declared because this window accepts events. The values of the terminals have been chosen to be mnemonic and have to match the values in the terminal mapping routine in pfig.cc, the main module. There are also private variables for saving state between predictions of non-terminals from this module.

```
grammar

create
        Window          w;
        =>
        $(
                canvas_swfd = canvas_pixwin = w
                        = xi->create_enable_context(@.ps);
                cerr << "Canvas window " << hex(w) << "\n";
                canvas_leftbut_proc = canvas_middlebut_proc =
                        canvas_locmove_proc = null_proc;
                canvas_kbd_proc = null_proc;
                menu_count = @.ni;
                m_info = @.Ms;
        $)
        M:create(@.Ms,@.is,@.ni)
        $(
                mctx = @M:create.ctx1;
                hctx = @.hctx;
```

```
                vctx = @.vctx;
        $)
        returns (w);

destroy =>
        $(
                cerr << "Canvas destroy\n";
        $);
```

The function of this `create` non-terminal is similar to that in the wrapper module: obtain a context, initialize variables, predict the `create` non-terminal of a child and return the new context to its parent.

```
S       &>
        repaint_loop work_loop;

repaint_loop    =>
        repaint*;

work_loop       =>
        work*;
```

The S non-terminal starts two sub-parsers in parallel, one to deal with exposure events and the other to deal with user input. This arrangement allows the user input productions to be changed without affecting exposure handling.

```
repaint =>
        expose
        $(
                cerr << "Canvas expose ctx " << hex(@.ctx) << "\n";
                redisplay_canvas();
        $);

work
        XButtonEvent    *e;
        =>
        left
        $(
                cerr << "Canvas left button ctx " << hex(@.ctx) << "\n";
                e = (XButtonEvent *)xi->xevent();
                (*canvas_leftbut_proc)(e->x, e->y);
        $);

work
        XButtonEvent    *e;
```

```
        =>
        middle
        $(
                cerr << "Canvas middle button ctx " << hex(@.ctx) << "\n";
                e = (XButtonEvent *)xi->xevent();
                (*canvas_middlebut_proc)(e->x, e->y);
        $);

work    =>
        right
        $(
                cerr << "Canvas right button ctx " << hex(@.ctx) << "\n";
                xi->enable_context(m_info);
        $)
        M:S(menu_count,mctx);

work
        XAnyEvent       *e;
        =>
        pointer
        $(
                e = xi->xevent();
                pointer_move(e);
                xi->change_context(e, (Window)hctx);
                xi->send_output(0, e);
                // one ruler handles both
        $);

work
        XAnyEvent       *e;
        =>
        key
        $(
                e = xi->xevent();
                key_press(e);
        $);
```

All that these productions have to do is invoke the appropriate semantic routines. The production that handles pointer motion is worth commenting on: it changes the context of the token and forwards it to a ruler window. This useful device is explained in Chapter 7.

Button module

A button provides an area that can be clicked upon to invoke an action routine. Although there is but one template, many instances of the button module are created because `pfig` has a panel of buttons. The panes in the pop-up menu are also instances of buttons.

```
// button window for pfig
module pbutton

$(
#include        <std.h>
#include        <stream.h>
#include        <Xinput.h>
#include        "pfig.h"
$)

        include "pbutton.pi";
        terminal expose = 'e',
                left = 'l', pointerin = 'f', pointerout = 'F';
        nonterm work, repaint;
        nonterm click, events;

private
        Wstruct *self;
        int     number;
```

The preliminaries are similar to the other modules, with the one difference that buttons are sensitive to the cursor moving in and out of them so that they can draw attention to themselves when they are prepared to accept input.

```
grammar

create
        Window          w;
        =>
        $(
                number = @.n;
                w = xi->create_enable_context(self = @.ps);
                cerr << "Button window " << hex(w) << "\n";
        $)
        returns (w);

destroy =>
        $(
                cerr << "Button destroy\n";
        $);
```

The create production obtains a context, saves information in private variables and returns the new context.

```
S       &>
```

```
        work
        repaint;

work    =>
        click*;

repaint =>
        events*;
```

Here we create two sub-parsers to deal with user input and exposure events.

```
click
        XAnyEvent       *e;
        =>
        left
        $(
                // cerr << "Button left ctx " << hex(@.ctx) << "\n";
                xi->refresh_contents(self);
                // monkey with button event and pretend it came from panel
                e = xi->xevent();
                xi->change_context(e, self->parent);
                // this is terrible, but to do it right is messy
                // put button number in last long of packet
                spare_slot(e) = number;
                xi->send_output(0, e);
        $);

events  =>
        expose
        $(
                // cerr << "Button expose ctx " << hex(@.ctx) << "\n";
                xi->refresh_contents(self);
        $);

events  =>
        pointerin
        $(
                cerr << "Button pointerin ctx " << hex(@.ctx) << "\n";
                xi->highlight_window(self);
                xi->refresh_contents(self);
        $);

events  =>
        pointerout
        $(
                cerr << "Button pointerout ctx " << hex(@.ctx) << "\n";
                xi->unhighlight_window(self);
                xi->refresh_contents(self);
        $);
```

The `click` production forwards the click event to the panel window. Each button window has a unique index assigned via an attribute passed in at creation time. This index is forwarded to the panel in the event record so that the panel knows which button the click originated from.

Panel module

The panel module implements a bank of switches. It encloses many instances of buttons.

```
// panel window for pfig
module ppanel

$(
#include        <std.h>
#include        <stream.h>
#include        <Xinput.h>
#include        "pfig.h"
#include        "switch.h"
$)

        include "ppanel.pi";
        terminal expose = 'e',
                left = 'l';
        nonterm repaint, work;
        nonterm spawn;
        nonterm click, expose_loop;

external pbutton
        include "pbutton.pi";

private
        module pbutton handle;
        Wstruct *win;
        Wstruct *button_info;
        int nsw;
$(
        extern "C" void do_switch(int);
$)

grammar

create
        Window          w;
        Wstruct         *b;
        module pbutton  B;
        int             i;
        =>
        $(
```

```
		button_info = @.bs;
		nsw = @.nb;
		w = xi->create_enable_context(@.ps);
		cerr << "Panel window " << hex(w) << "\n";
		for (i = 0; i < @.nb; ++i)
		{
			b = &button_info[i];
			b->dpy = @.ps->dpy;
			b->parent = w;
			b->fgc = @.ps->fgc;	/* inherit from panel */
			b->bgc = @.ps->bgc;
			b->eventmask = ButtonPressMask | ExposureMask;
	$)
			B:create(b,i)
	$(
			b->pg_mod = B;
		}
	$)
	returns (w);

destroy =>
	$(
		cerr << "Panel destroy\n";
	$);
```

The number of buttons in the panel is passed in an inherited attribute from the wrapper module. PENGUIN does not have a construct to predict a large but finite number of external non-terminals so an escape to the host language is used. The `for` loop initializes each button's configuration structure and then predicts the `start` symbol in that button.

```
S	&>
	repaint work;

repaint =>
	expose_loop*;

expose_loop	=>
	expose
	$(
		cerr << "Panel expose ctx " << hex(@.ctx) << "\n";
	$);

work	=>
	$(
		register int		i;
```

```
        for (i = 0; i < nsw; ++i)
        {
                win = &button_info[i];
                handle = (pbutton *)win->pg_mod;
$)
                spawn
$(
        }
$)
click+;

spawn   &:
        handle:S(win->self);
```

Here as before, the **for** loop escape to the host language is used to fork all the sub-parsers for the buttons in parallel.

```
click
        int             i;
        XAnyEvent       *e;
        XButtonEvent    *b;
        F_switch        *sw;
        F_switch        *old;
        =>
        left
        $(
                e = xi->xevent();
                b = (XButtonEvent *)e;
                cerr << "Panel left button ctx " << hex(@.ctx) << "\n";
                cerr << "Button " << (i = spare_slot(e)) << " Coordinates "
                        << b->x << " " << b->y << " "
                        << b->x_root << " " << b->y_root << "\n";
                do_switch(i);
        $);
```

Message module

The message window displays prompts and diagnostics from the program.

```
// message window for pfig
module pmessage

$(
#include        <std.h>
#include        <stream.h>
#include        <Xinput.h>
#include        "pfig.h"
```

```
#include        "resources.h"
$)

        include "pmessage.pi";
        terminal msg = '1', expose = 'e';
        nonterm repaint, message;
        nonterm repaint_loop, message_loop;

private
        Wstruct *self;

grammar

create
        Window          w;
        =>
        $(
                msgswfd = msg_pixwin = w
                        = xi->create_enable_context(self = @.ps);
                cerr << "Message window " << hex(w) << "\n";
        $)
        returns (w);

destroy =>
        $(
                cerr << "Message destroy\n";
        $);

S       &>
        repaint_loop message_loop;

repaint_loop    =>
        repaint*;

message_loop    =>
        message*;

repaint =>
        expose
        $(
                cerr << "Message expose ctx " << hex(@.ctx) << "\n";
                xi->refresh_contents(self);
        $);

message
        XAnyEvent       *e;
        =>
        msg
        $(
                e = xi->xevent();
                self->legend = (char *)spare_slot(e);
```

```
		xi->refresh_contents(self);
	$);
```

Two sub-parsers handle exposures and incoming messages. The message module assumes that a modules wishing to display a message places a pointer to the null-terminated string in a spare field of the message token and sends this token to the message window. The message window changes the field of the window structure that specifies the window legend and then calls the run time library routine to refresh the window.

Ruler module

One ruler module serves for both the vertical and horizontal rulers since both of them respond to the same events. Here, as for buttons, the ability to create multiple instances of a module is essential.

```
// ruler window for pfig
module pruler

$(
#include	<std.h>
#include	<stream.h>
#include	<Xinput.h>
#include	"pfig.h"
$)

	include "pruler.pi";
	terminal expose = 'e', pointer = 'p';
	nonterm repaint_loop, work_loop, repaint, work;

private
	Wstruct *self;
$(
	extern "C" void set_rulermark(int, int);
$)

grammar

create
	Window		w;
	=>
	$(
		w = xi->create_enable_context(self = @.ps);
		cerr << "Ruler window " << hex(w) << "\n";
	$)
	returns (w);
```

```
destroy =>
        $(
                cerr << "Ruler destroy\n";
        $);

S       &>
        repaint_loop work_loop;

repaint_loop    =>
        repaint*;

repaint =>
        expose
        $(
                cerr << "Ruler expose ctx " << hex(@.ctx) << "\n";
                xi->refresh_contents(self);
        $);

work_loop       =>
        work*;

work
        XMotionEvent    *e;
        =>
        pointer
        $(
                // cerr << "Ruler motion ctx " << hex(@.ctx) << "\n";
                e = (XMotionEvent *)xi->xevent();
                set_rulermark(e->x, e->y);
        $);
```

As pointed out in the discussion of the canvas window, the ruler windows do not receive user input directly but have motion events forwarded to them from the canvas area.

Menu module

The menu module handles the pop-up menu which is activated by clicking on the right mouse button.

```
// menu window for pfig
module pmenu

$(
#include        <std.h>
#include        <stream.h>
#include        <Xinput.h>
#include        "pfig.h"
```

```
$)

        include "pmenu.pi";
        terminal expose = 'e',
                left = 'l';
        nonterm exposures, click;
        nonterm repaint, work(int ni);
        nonterm spawn;

external pbutton
        include "pbutton.pi";

private
        module pbutton handle;
        Wstruct *item_info;
        Wstruct *win;
$(
        extern "C" void pane_select(int);
$)

grammar

create
        Window          w;
        Wstruct         *b;
        module pbutton  B;
        int             i;
        =>
        $(
                b = item_info = @.is;
                @.ps->height = (b->height + b->bwidth * 2) * @.ni;
                w = xi->create_context(@.ps);
                cerr << "Menu window " << hex(w) << "\n";
                for (i = 0; i < @.ni; ++i, ++b)
                {
                        b->dpy = @.ps->dpy;
                        b->parent = w;
                        b->hoff = 0;
                        b->voff = ((b->bwidth * 2 + b->height) * i);
                        b->fgc = @.ps->fgc;     /* inherit from menu */
                        b->bgc = @.ps->bgc;
                        b->fontname = @.ps->fontname;
                        b->eventmask = EnterWindowMask | LeaveWindowMask |
                                ExposureMask | ButtonPressMask;
        $)
                        B:create(b,i)
        $(
                        b->pg_mod = B;
                }
        $)
        returns (w);
```

```
destroy =>
        $(
                cerr << "Menu destroy\n";
        $);

S       &>
        repaint work(@.ni);

repaint =>
        exposures*;

exposures       =>
        expose
        $(
                cerr << "Menu expose ctx " << hex(@.ctx) << "\n";
        $);

work    =>
        $(
                register int            i;

                for (i = 0; i < @.ni; ++i)
                {
                        win = &item_info[i];
                        handle = (pbutton *)win->pg_mod;
        $)
                        spawn
        $(
                }
        $)
        click+;

spawn   &:
        handle:S(win->self);

click
        int             i;
        XAnyEvent       *e;
        XButtonEvent    *b;
        =>
        l-ft
        $(
                e = xi->xevent();
                b = (XButtonEvent *)e;
                cerr << "Menu left button ctx " << hex(@.ctx) << "\n";
                cerr << "Panel " << (i = spare_slot(e)) << "\n";
                pane_select(i);
        $);
```

The menu module is very similar to the panel window. One difference is that its parent window (in X11 terms) is not the wrapper window but the background window of the screen. This is because the menu is a pop-up window that should not be clipped by the wrapper area. Again, each pane in the menu is assigned a unique index which is used to identify the origin of a click forwarded from some button.

B.2 Main program and auxiliary routines

Here is the main routine of the pfig program. It is written in C++ because it does not deal with events and it requires actions that cannot be expressed with PENGUIN code.

```
#include        <stream.h>
#include        <pglib.h>
#include        <Xinput.h>

#include        "xpfig.h"
#include        "pfig.h"
#include        "version.h"
#include        "pwrapper.h"
#include        "resources.h"
#include        "switch.h"

int event_translate(XAnyEvent *e)
/*
**      Translate an X event into a flat alphabet
*/
{
        switch (e->type)
        {
        case KeyPress:
                return ('k');
        case ButtonPress:
                switch (((XButtonEvent *)e)->button)
                {
                case Button1:
                        return ('l');
                case Button2:
                        return ('m');
                case Button3:
                        return ('r');
                }
        case EnterNotify:
                return('f');
        case LeaveNotify:
                return('F');
```

```
        case Expose:
                return ('e');
        case MotionNotify:
                return ('p');
        }
        return ('?');
}
```

The event_translate routine is called by the run time library to examine the input token and return the appropriate value to the parser. Here it is used to flatten the token record. Left, middle and right buttons can be distinguished only by examining another field of the token record. This routine returns distinct values for each button.

```
Wstruct         wrapper_info =
{
        WRAPPER_WIDTH, WRAPPER_HEIGHT, WRAPPER_BORDER,
        0, 0,
        0, 0, 0,
        0, 1,                   /* fg and bg colours, will be pgmed */
        0, 0,                   /* normal and highlighted gcs */
        0, 0, 0, 0,             /* state */
        0, 0, 0,
        0
};
Wstruct         panel_info =
{
        PANEL_WIDTH, PANEL_HEIGHT, PANEL_BORDER,
        PANEL_HOFF, PANEL_VOFF,
        0, 0, 0,
        0, 1,                   /* fg and bg colours, will be pgmed */
        0, 0,                   /* normal and highlighted gcs */
        0, 0, 0, 0,             /* state */
        0, 0, 0,
        0
};
#define         SWITCH_COUNT    42      /* should match panel.c */
Wstruct         button_info[SWITCH_COUNT];
Wstruct         canvas_info =
{
        CANVAS_WIDTH, CANVAS_HEIGHT, CANVAS_BORDER,
        CANVAS_HOFF, CANVAS_VOFF,
        0, 0, 0,
        0, 1,                   /* fg and bg colours, will be pgmed */
        0, 0,                   /* normal and highlighted gcs */
        0, 0, 0, 0,             /* state */
        0, 0, 0,
        KeyPressMask|ButtonPressMask|ExposureMask|PointerMotionMask
```

```
};
Wstruct         hruler_info =
{
        HRULER_WIDTH, HRULER_HEIGHT, HRULER_BORDER,
        HRULER_HOFF, HRULER_VOFF,
        0, 0, 0,
        0, 1,                   /* fg and bg colours, will be pgmed */
        0, 0,                   /* normal and highlighted gcs */
        0, 0, 0, 0,             /* state */
        0, 0, 0,
        ExposureMask
};
Wstruct         vruler_info =
{
        VRULER_WIDTH, VRULER_HEIGHT, VRULER_BORDER,
        VRULER_HOFF, VRULER_VOFF,
        0, 0, 0,
        0, 1,                   /* fg and bg colours, will be pgmed */
        0, 0,                   /* normal and highlighted gcs */
        0, 0, 0, 0,             /* state */
        0, 0, 0,
        ExposureMask
};
Wstruct         message_info =
{
        MESSAGE_WIDTH, MESSAGE_HEIGHT, MESSAGE_BORDER,
        MESSAGE_HOFF, MESSAGE_VOFF,
        0, 0, 0,
        0, 1,                   /* fg and bg colours, will be pgmed */
        0, 0,                   /* normal and highlighted gcs */
        0, 0, 0, 0,             /* state */
        "fixed", 0, 0,
        ExposureMask            /* however, it gets sent fake key events */
};
Wstruct         menu_info =
{
        MENU_WIDTH, MENU_HEIGHT, MENU_BORDER,
        MENU_HOFF, MENU_VOFF,
        0, 0, 0,
        0, 1,                   /* fg and bg colours, will be pgmed */
        0, 0,                   /* normal and highlighted gcs */
        0, 0, 0, 0,             /* state */
        "fixed", 0, 0,
        0
};
Wstruct         item_info[ITEM_COUNT];
```

The preceding window structures describe all the interesting attributes of the windows managed by the preceding modules. These structure initializations specify the physical position, size, text font used, if any, and the set of events that

each window responds to. Pointers to these structures are passed to the modules , attributes and thence to the library routines that interface to the presentation component.

Note that forwarded events cannot be stopped by excluding the event from event set in the window structure. Only the presentation component (X server) obeys the event set. Forwarded events are always delivered, along with events from the presentation, if requested in the event set.

```
Xinput          *xi;

static void send_message(char *m)
{
        register XAnyEvent      *last_event = xi->xevent();

        xi->change_context(last_event, message_info.self);
        spare_slot(last_event) = (long)m;
        xi->send_event(last_event);
}

static void send_button(int i, int etype)
{
        register XAnyEvent      *last_event = xi->xevent();
        XEvent                  e = *((XEvent *)last_event);

        e.type = etype;
        xi->change_context((XAnyEvent *)&e, button_info[i].self);
        xi->send_event((XAnyEvent *)&e);
}

extern  F_switch        switches[];
extern  void            (*sendmsg)();
extern  void            (*sendsw)();

main(int argc, char *argv[])
{
        register int            i;
        register Wstruct        *s;
        extern MENUITEM         pumenu_items[];
        pwrapper                w;
        _pwrapper_create_attr   wc_attr;
        _pwrapper_S_attr        wS_attr;
        subparser               sp;
        XEvent                  e;

        cerr << "Initializing FIG " << FIG_VERSION << MINOR_VERSION <<
                " (PenGuin) ... \n";

        ip = xi = new Xinput(0, event_translate);       // needed by pglib
```

The input routines form a C++ class so we need to create one instance. Although there is never any need for more than one instance at the moment, classes are useful for hiding information.

```
sendmsg = (void (*)())send_message;
sendsw = (void (*)())send_button;
// fill in display info and root window id
menu_info.dpy = message_info.dpy =
        hruler_info.dpy = vruler_info.dpy = canvas_info.dpy =
        panel_info.dpy = wrapper_info.dpy = xi->Dpy();
menu_info.parent = wrapper_info.parent =
        RootWindow(wrapper_info.dpy, 0);

// create the windows
wrapper_info.fgc = BlackPixel(wrapper_info.dpy, 0);
wrapper_info.bgc = WhitePixel(wrapper_info.dpy, 0);
wc_attr.wrapper_str = &wrapper_info;
panel_info.fgc = BlackPixel(panel_info.dpy, 0);
panel_info.bgc = WhitePixel(panel_info.dpy, 0);
wc_attr.panel_str = &panel_info;
canvas_info.fgc = BlackPixel(canvas_info.dpy, 0);
canvas_info.bgc = WhitePixel(canvas_info.dpy, 0);
for (s = button_info, i = 0; i < SWITCH_COUNT; ++s, ++i)
{
        F_switch        *x;
        x = &switches[i];
        s->bwidth = SWITCH_BORDER;
        s->width = SWITCH_WIDTH;
        s->height = SWITCH_HEIGHT;
        s->hoff = SWITCH_SPACING +
        (SWITCH_BORDER*2+SWITCH_WIDTH+SWITCH_SPACING) * x->x;
        s->voff = SWITCH_SPACING +
        (SWITCH_BORDER*2+SWITCH_HEIGHT+SWITCH_SPACING) * x->y;
        s->bgdimage = x->icon;
        s->state = 0;
        s->legend = 0;
}
wc_attr.button_str = button_info;
wc_attr.nbuttons = SWITCH_COUNT;
wc_attr.canvas_str = &canvas_info;
hruler_info.fgc = BlackPixel(hruler_info.dpy, 0);
hruler_info.bgc = WhitePixel(hruler_info.dpy, 0);
wc_attr.hruler_str = &hruler_info;
vruler_info.fgc = BlackPixel(vruler_info.dpy, 0);
vruler_info.bgc = WhitePixel(vruler_info.dpy, 0);
wc_attr.vruler_str = &vruler_info;
message_info.fgc = BlackPixel(message_info.dpy, 0);
message_info.bgc = WhitePixel(message_info.dpy, 0);
wc_attr.message_str = &message_info;
```

```
menu_info.fgc = BlackPixel(menu_info.dpy, 0);
menu_info.bgc = WhitePixel(menu_info.dpy, 0);
wc_attr.menu_str = &menu_info;
for (s = item_info, i = 0; i < ITEM_COUNT; ++i, ++s)
{
        s->bwidth = ITEM_BORDER;
        s->width = ITEM_WIDTH;
        s->height = ITEM_HEIGHT;
        s->bgdimage = 0;
        s->state = 0;
        s->legend = pumenu_items[i].label;
}
wc_attr.item_str = item_info;
wc_attr.nitems = ITEM_COUNT;
```

The tedious code above initializes fields in the various window configuration structures. Among other things, the spacings for the panel buttons are calculated here. This code really should belong in the panel module.

```
compatibility(xi->Dpy(), BlackPixel(wrapper_info.dpy, 0),
        WhitePixel(wrapper_info.dpy, 0));
w.create(&sp, wc_attr);
more_compatibility(xi->Dpy(), BlackPixel(wrapper_info.dpy, 0),
        WhitePixel(wrapper_info.dpy, 0));
```

The first phase creates all the contexts needed. The other routines contain kludgery to make a C++ program work with C routines.

```
pg_init(&sp, &w, 0, 6, wc_attr.ctx1, wc_attr.ctx2, wc_attr.ctx3,
        wc_attr.ctx4, wc_attr.ctx5, wc_attr.ctx6);
```

This call to `pg_init` loads the initial set of valid contexts into the context dictionary. Henceforth the dictionary is updated by forks and joins.

```
// start the pwrapper window
wS_attr.dpy = wrapper_info.dpy;
wS_attr.ctx = wc_attr.ctx1;
wS_attr.ctx1 = wc_attr.ctx2;
wS_attr.nbuttons = SWITCH_COUNT;
wS_attr.ctx2 = wc_attr.ctx3;
wS_attr.ctx3 = wc_attr.ctx4;
wS_attr.ctx4 = wc_attr.ctx5;
wS_attr.ctx5 = wc_attr.ctx6;
w.S(&sp, wS_attr);
```

This tedious code to copy contexts is forced by the current implementation of attribute passing. The call to `w.S` starts everything going.

```
        // should never be called: catch last event
        XNextEvent(wrapper_info.dpy, &e);
        exit(0);
}
```

It should be possible to move much of the initialization code into the a run time library module as the creation and execution phases are common to all PENGUIN applications.

B.3 Example output from PENGUIN compiler

Here is the C++ code automatically generated by PENGUIN compiler from the PENGUIN code for canvas module.

```
#include        <std.h>
#include        <stream.h>
#include        <Xinput.h>
#include        "pfig.h"
#include        "resources.h"

#include        "pcanvas.h"
#include        "pmenu.h"

        extern "C" void redisplay_canvas();
        extern "C" void pointer_move();
        extern "C" void key_press();
        extern "C" void null_proc();
        extern void (*canvas_leftbut_proc)(int, int);
        extern void (*canvas_middlebut_proc)(int, int);
        extern void (*canvas_locmove_proc)(int, int);
        extern void (*canvas_kbd_proc)(int, int);

static transentry       _transtab[115] =
{
/*0*/   -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
/*10*/  -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
/*20*/  -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
/*30*/  -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
/*40*/  -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
/*50*/  -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
/*60*/  -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
/*70*/  -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
```

```
/*80*/  -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
/*90*/  -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
/*100*/ -1, 0, -1, -1, -1, -1, -1, 1, 2, 3,
/*110*/ -1, -1, 5, -1, 4,
};

static splitentry       _S_split[] =
{
        "\x080",
        "\x07c",
        0,
};

static predpair _predlist[] =
{
        { 0, offsetof(_pcanvas_repaint_attr,ctx) },
        { -1, 0 },
        { 5, offsetof(_pcanvas_work_attr,ctx) },
        { -1, 0 },
        { 1, offsetof(_pcanvas_work_attr,ctx) },
        { -1, 0 },
        { 2, offsetof(_pcanvas_work_attr,ctx) },
        { -1, 0 },
        { 3, offsetof(_pcanvas_work_attr,ctx) },
        { -1, 0 },
        { 4, offsetof(_pcanvas_work_attr,ctx) },
        { -1, 0 },
};

static predentry        _predtab[2][7] =
{
/*0*/   &_predlist[0],0,0,0,0,0,0,
/*1*/   0,&_predlist[2],&_predlist[4],&_predlist[6],&_predlist[8],
&_predlist[10],0,
};

pcanvas::pcanvas()
{
        name = "pcanvas";
        maxvalue = 114;
        nterms = 2; terms = 7;
        nprods = 6; nsprods = 5;
        transtab = _transtab,
        predtab = &_predtab[0][0];
        M = 0;
}

void pcanvas::repaint(subparser *sp, _pcanvas_repaint_attr& _l)
{
        switch(pg_decide(sp, 0/*repaint*/, (char *)&_l))
        {
```

```
        case 0: repaint_000(sp, _1); break;
        default: pg_error(); break;
        }
}

void pcanvas::work(subparser *sp, _pcanvas_work_attr& _1)
{
        switch(pg_decide(sp, 1/*work*/, (char *)&_1))
        {
        case 1: work_001(sp, _1); break;
        case 2: work_002(sp, _1); break;
        case 3: work_003(sp, _1); break;
        case 4: work_004(sp, _1); break;
        case 5: work_005(sp, _1); break;
        default: pg_error(); break;
        }
}

void pcanvas::repaint_000(subparser *sp, _pcanvas_repaint_attr& _1)
{


        pg_expect(sp,0/*expose*/,_1.ctx);
        cerr << "Canvas expose ctx " << hex(_1.ctx) << "\n";
                redisplay_canvas();

}

void pcanvas::work_001(subparser *sp, _pcanvas_work_attr& _1)
{
        XButtonEvent *e;


        pg_expect(sp,2/*left*/,_1.ctx);
        cerr << "Canvas left button ctx " << hex(_1.ctx) << "\n";
                e = (XButtonEvent *)xi->xevent();
                (*canvas_leftbut_proc)(e->x, e->y);

}

void pcanvas::work_002(subparser *sp, _pcanvas_work_attr& _1)
{
        XButtonEvent *e;


        pg_expect(sp,3/*middle*/,_1.ctx);
        cerr << "Canvas middle button ctx " << hex(_1.ctx) << "\n";
                e = (XButtonEvent *)xi->xevent();
                (*canvas_middlebut_proc)(e->x, e->y);

}
```

```
void pcanvas::work_003(subparser *sp, _pcanvas_work_attr& _1)
{
        _pmenu_S_attr   _2;


        pg_expect(sp,4/*right*/,_1.ctx);
        cerr << "Canvas right button ctx " << hex(_1.ctx) << "\n";
                xi->enable_context(m_info);

        _2.ni = menu_count;
        _2.ctx = mctx;
        sp->mod = M;
        M->S(sp, _2);
        sp->mod = this;
}

void pcanvas::work_004(subparser *sp, _pcanvas_work_attr& _1)
{
        XAnyEvent *e;


        pg_expect(sp,5/*pointer*/,_1.ctx);
                e = xi->xevent();
                pointer_move(e);
                xi->change_context(e, (Window)hctx);
                xi->send_output(0, e);
                // one ruler handles both

}

void pcanvas::work_005(subparser *sp, _pcanvas_work_attr& _1)
{
        XAnyEven  *e;


        pg_expect(sp,1/*key*/,_1.ctx);
                e = xi->xevent();
                key_press(e);

}

void pcanvas::create(subparser *sp, _pcanvas_create_attr& _1)
{
        _pmenu_create_attr      _1;
        Window w;


                canvas_swfd = canvas_pixwin = w
                        = xi->create_enable_context(_1.ps);
                cerr << "Canvas window " << hex(w) << "\n";
```

```
                canvas_leftbut_proc = canvas_middlebut_proc =
                        canvas_locmove_proc = null_proc;
                canvas_kbd_proc = null_proc;
                menu_count = _l.ni;
                m_info = _l.Ms;

        _1.ps = _l.Ms;
        _1.is = _l.is;
        _1.ni = _l.ni;
        _1.ctx = _l.ctx;
        M = new pmenu;
        sp->mod = M;
        M->create(sp, _1);
        sp->mod = this;
                mctx = _1.ctx1;
                hctx = _l.hctx;
                vctx = _l.vctx;

        _l.ctx1 = w;
}

void pcanvas::destroy(subparser *sp, _pcanvas_destroy_attr& _l)
{


                cerr << "Canvas destroy\n";

}

void pcanvas::S(subparser *sp, _pcanvas_S_attr& _l)
{
        _pcanvas_repaint_loop_attr&     _0 = *new
                _pcanvas_repaint_loop_attr;
        _pcanvas_work_loop_attr&        _1 = *new
                _pcanvas_work_loop_attr;

        _0.ctx = _l.ctx;
        _1.ctx = _l.ctx;
        register branch *s;
        register connector      *c = conalloc(2);
        c->coninit(2,2,sp);
        // Initialize connector element 0
        s = &c->branches[0];
        s->brinit((mproc)repaint_loop,&_0);
        s->spinit(sp->mod,1,&_0.ctx,(unsigned char *)_S_split[0]);
        // Initialize connector element 1
        s = &c->branches[1];
        s->brinit((mproc)work_loop,&_1);
        s->spinit(sp->mod,1,&_1.ctx,(unsigned char *)_S_split[1]);
        pg_fork(sp, c);
}
```

```
void pcanvas::repaint_loop(subparser *sp,
        _pcanvas_repaint_loop_attr& _1)
{
        _pcanvas_repaint_attr   _0;

        static predpair _0_flist[] = {
        { 0,offsetof(_pcanvas_repaint_attr,ctx) },
        { -1, 0 },
        };
        static predentry        _0_ftab[] = {
        &_0_flist[0],
        0,
        0,
        0,
        0,
        0,
        0,
        };

        _0.ctx = _1.ctx;
        while (pg_extend(sp,_0_ftab,(char *)&_1) >= 0)
                repaint(sp, _0);
}

void pcanvas::work_loop(subparser *sp, _pcanvas_work_loop_attr& _1)
{
        _pcanvas_work_attr      _0;

        static predpair _0_flist[] = {
        { 5,offsetof(_pcanvas_work_attr,ctx) },
        { -1, 0 },
        { 1,offsetof(_pcanvas_work_attr,ctx) },
        { -1, 0 },
        { 2,offsetof(_pcanvas_work_attr,ctx) },
        { -1, 0 },
        { 3,offsetof(_pcanvas_work_attr,ctx) },
        { -1, 0 },
        { 4,offsetof(_pcanvas_work_attr,ctx) },
        { -1, 0 },
        };
        static predentry        _0_ftab[] = {
        0,
        &_0_flist[0],
        &_0_flist[2],
        &_0_flist[4],
        &_0_flist[6],
        &_0_flist[8],
        0,
        };
```

```
        _0.ctx = _1.ctx;
        while (pg_extend(sp,_0_ftab,(char *)&_1) >= 0)
                work(sp, _0);
}
```

Bibliography

[Aho *et al.*, 1986] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers — Principles, Techniques and Tools*, Addison-Wesley, 1986.

[Apple Computers, 1986] Apple Computers, *Using the Macintosh Plus*, 1986.

[Bertin, 1983] Jacques Bertin, *Semiology of Graphics*, The University of Wisconsin Press, Madison, Wisconsin 53715, 1983.

[Donnelly and Stallman, 1988] Charles Donnelly and Richard Stallman, *BISON: The YACC-compatible Parser Generator*, 1988.

[Edmonds, 1981] E. A. Edmonds, *Adaptive Man-Computer Interfaces*, pages 389-426, Computing Skills and the User Interface. Academic Press, London, 1981.

[Flecchia and Bergeron, 1987] Mark A. Flecchia and Daniel R. Bergeron, "Specifying Complex Dialogs in ALGAE," In *Conference Proceedings of Human Factors in Computing Systems and Graphics Interface, Toronto, Canada*, April 1987.

[Foley and Wallace, 1974] James D. Foley and Victor L. Wallace, "The Art of Natural Graphic Man-Machine Conversation," *Proceedings of the IEEE*, 62(4):462–471, April 1974.

[Green, 1985] Mark Green, "The University of Alberta User Interface Management System," *Computer Graphics*, July 1985.

[Guest, 1982] Stephen P. Guest, "The Use of Software Tools for Dialogue Design," *International Journal of Man-Machine Studies*, 16:263–285, 1982.

[Haeberli, 1988] Paul E. Haeberli, "ConMan: A visual programming language for interactive graphics," In *SIGGRAPH '88 Conference Proceedings*, pages 103-111, August 1988.

[Hanau and Lenorovitz, 1980] Paul. R. Hanau and David. R. Lenorovitz, "Prototyping and Simulation Tools for User/Computer Dialogue Design," In *SIGGRAPH '80 Conference Proceedings*, pages 271-278, 1980.

[Ingalls *et al.*, 1988] Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle, "Fabrik: A Visual Programming Environment," In *OOPSLA '88 Conference Proceedings*, pages 176–190, September 1988.

[ISO/IEC, 1987] ISO/IEC, *Information Processing — Open Systems Interconnection — Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*, 1987, International Standard 8825.

[Jacob, 1983] Robert J. K. Jacob, "Using Formal Specifications in the Design of a Human-Computer Interface," *Communications of the ACM*, 26(4):259–264, April 1983.

[Jensen and Wirth, 1975] Kathleen Jensen and Niklaus Wirth, *Pascal User Manual and Report*, Springer-Verlag, New York, second edition, 1975.

[Kernighan and Ritchie, 1978] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1978.

[Knuth, 1968] Donald E. Knuth, "Semantics of Context-Free Languages," *Mathematical Systems Theory*, 2(2):127–145, June 1968.

[Lesk and Schmidt, 1979] M. E. Lesk and E. Schmidt, *Lex: A Lexical Analyser Generator*, 1979.

[Matthys, 1985] J. Matthys, "Recent Experiences with Input Handling at PMA," In *User Interface Management Systems*. Springer-Verlag, 1985.

[McCormack *et al.*, 1988] Joel McCormack, Paul Asente, and Ralph R. Swick, *X Toolkit Intrinsics—C Language X Interface*, 1988.

[McDonald and Dix, 1988] Chris McDonald and Trevor I. Dix, "Support for Graphs of Processes in a Command Interpreter," *Software Practice and Experience*, 18(10):1011–1016, October 1988.

[Meyer, 1988] Bertrand Meyer, *Object-oriented Software Construction*, Prentice-Hall, 1988.

[Microsystems, 1986] Sun Microsystems, *External Data Representation Protocol Specification*, Mountain View, California, February 1986, Part Number 800-1324-03.

[Morse, 1979] Alan Morse, "Some Principles for the Effective Display of Data," *Computer Graphics*, 13(2), August 1979.

[Motorola, Inc., 1985] Motorola, Inc., *MC68020 32-Bit Microprocessor User's Manual*, 2nd edition, 1985.

[Newman, 1968] W. M. Newman, "A System for Interactive Graphical Programming," In *SJCC 1968*, 1968.

[Olsen, 1983] Dan R. Olsen, Jr., "Automatic Generation of Interactive Systems," *Computer Graphics*, January 1983.

[Olsen and Dempsey, 1983] Dan R. Olsen, Jr. and Elizabeth P. Dempsey, "SYNGRAPH: A Graphical User Interface Generator," *Computer Graphics*, July 1983.

[Paxson, 1989] Vern Paxson, *Flex: Fast Lexical Analyser Genertor*, 1989.

[Pfaff, 1985] Günther E. Pfaff, editor, *User Interface Management Systems*, Springer-Verlag, 1985.

[Ritchie and Thompson, 1974] D. M. Ritchie and K. Thompson, "The UNIX Time Sharing System," *Communications of the ACM*, 17(7):365–375, July 1974.

[Scheifler and Gettys, 1987] Robert W. Scheifler and Jim Gettys, "The X Window System," *ACM Transactions on Graphics*, 6(2), April 1987.

[Scott and Finkel, 1988] M. L. Scott and A R. A. Finkel, "A Simple Mechanism for Type Security Across Compilation Units," *IEEE Transactions on Software Engineering*, 14(8):1238–1239, August 1988.

[Shneiderman, 1982] Ben Shneiderman, "Multiparty Grammars and Related Features for Defining Interactive Systems," *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-12(2):148–154, March 1982.

[Singh, 1985] Gurminder Singh, "Presentation Component for the U of Alberta UIMS," Master's thesis, University of Alberta, 1985.

[Stroustrup, 1986] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.

[Sun Microsystems, 1986] Sun Microsystems, *SunView System Programmer's Guide*, SMI, 1986.

[Thomas, 1983] James J. Thomas, "Graphical Input Interaction Technique (GIIT) Workshop Summary," *Computer Graphics*, January 1983.

[Tiemann, 1989] Michael D. Tiemann, *User's Guide to GNU C++*, 1989.

[van den Bos, 1979] Jan van den Bos, "Input Tools – A new language construct for input-driven programs," In *Proceedings of the European Conference on Applied Information Technology of IFIP*, September 1979.

[van den Bos, 1988] Jan van den Bos, "Abstract Interaction Tools: A Language for User Interface Management Systems," *ACM Transactions on Programming Languages and Systems*, 10(2):215–247, April 1988.

[Woods, 1980] W. A. Woods, "Transition Network Grammars for Natural Language Analysis," *Communications of the ACM*, 13(10):591–606, October 1980.

[Yourdon and Constantine, 1979] Edward Yourdon and Larry L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall, 1979.